Jupyter Notebooks

Madhavan Mukund

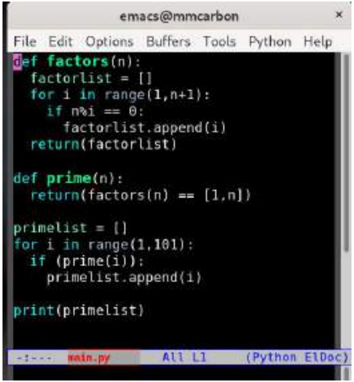
https://www.cmi.ac.in/~madhavan

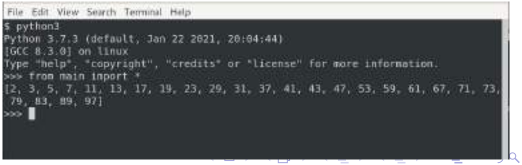
Programming, Data Structures and Algorithms using Python

Week 1

Writing and running code

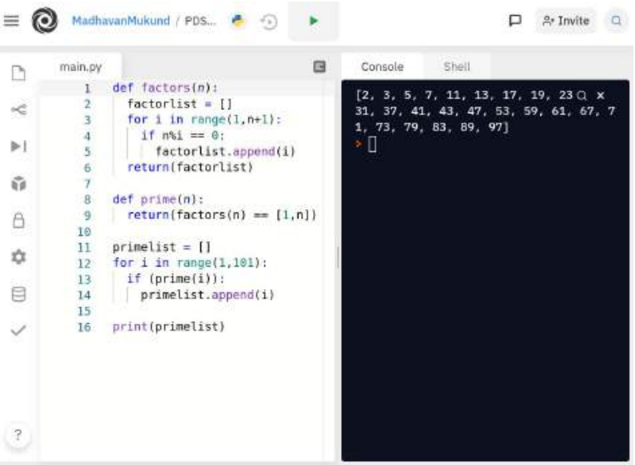
- Manual
  - Text editor to write code
  - Run at the command line





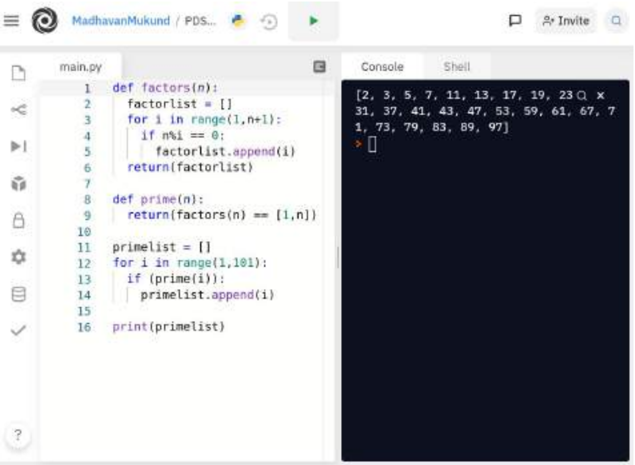
Writing and running code

- Manual
  - Text editor to write code
  - Run at the command line
- Integrated Development Environment (IDE)
  - Single application to write and run code
  - On desktop or online, `replit`
  - Quick update-run cycle
  - Debugging, testing, ...



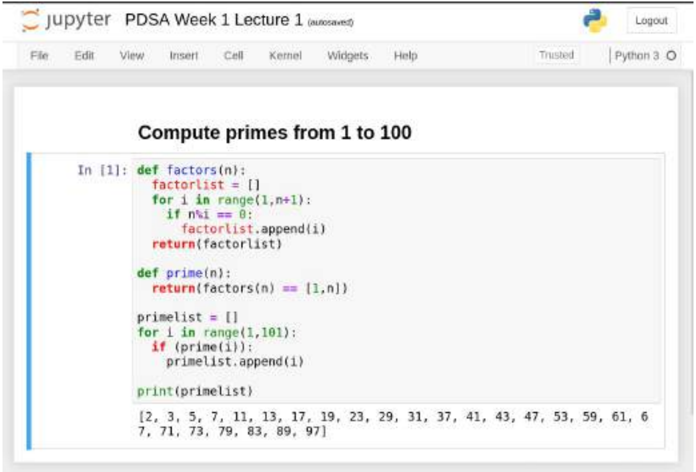
Writing and running code

- Manual
  - Text editor to write code
  - Run at the command line
- Integrated Development Environment (IDE)
  - Single application to write and run code
  - On desktop or online, replit
  - Quick update-run cycle
  - Debugging, testing, ...
- What more could one want?



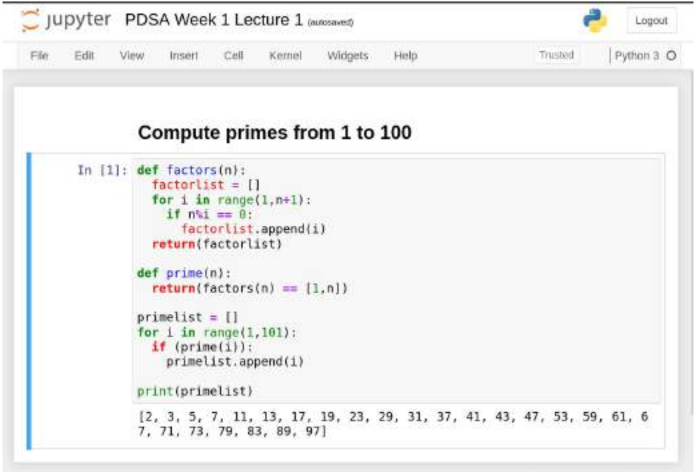
Collaboration

- Share your code
  - Collaborative development
  - Report your results



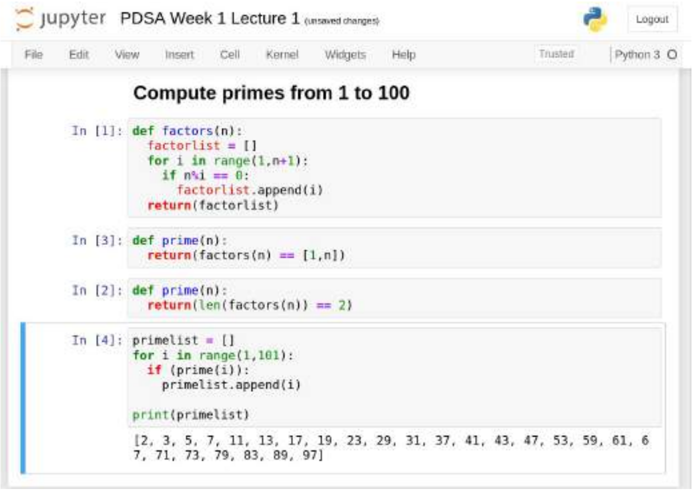
Collaboration

- Share your code
  - Collaborative development
  - Report your results
- Documentation
  - Interleave with the code



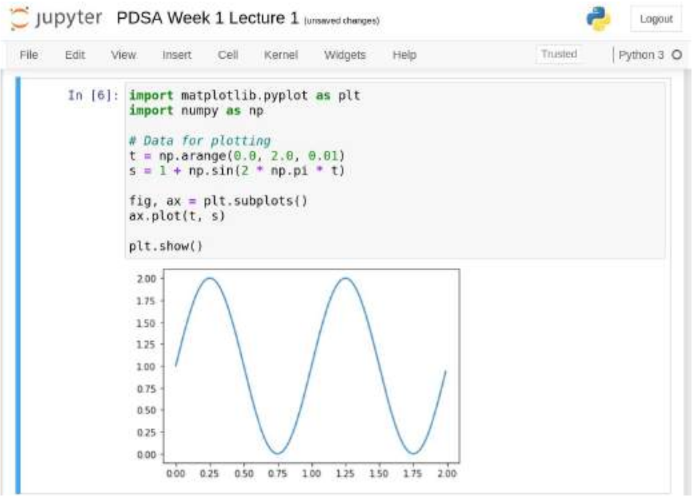
Collaboration

- Share your code
  - Collaborative development
  - Report your results
- Documentation
  - Interleave with the code
- Switch between different versions of code



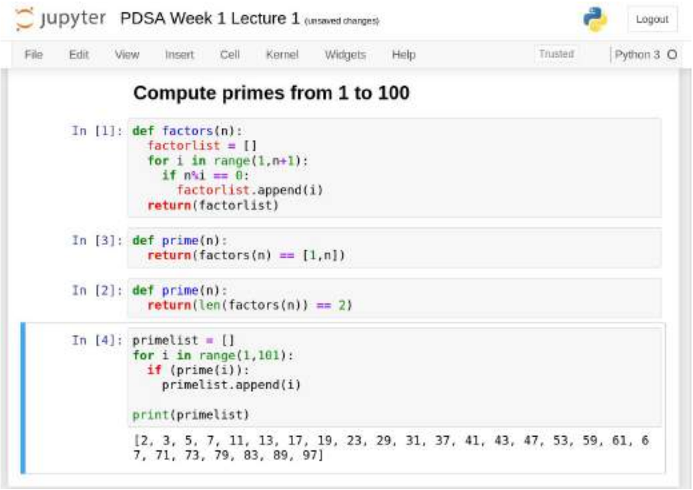
Collaboration

- Share your code
  - Collaborative development
  - Report your results
- Documentation
  - Interleave with the code
- Switch between different versions of code
- Export and import your project
- Preserve your output



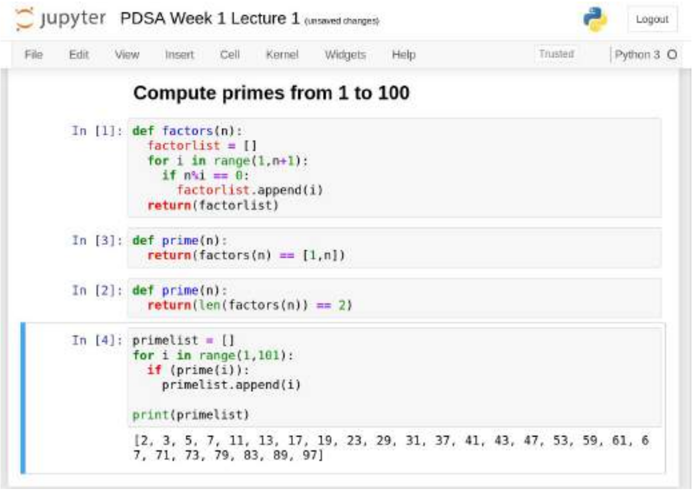
Jupyter notebook

- A sequence of cells
  - Like a one dimensional spreadsheet



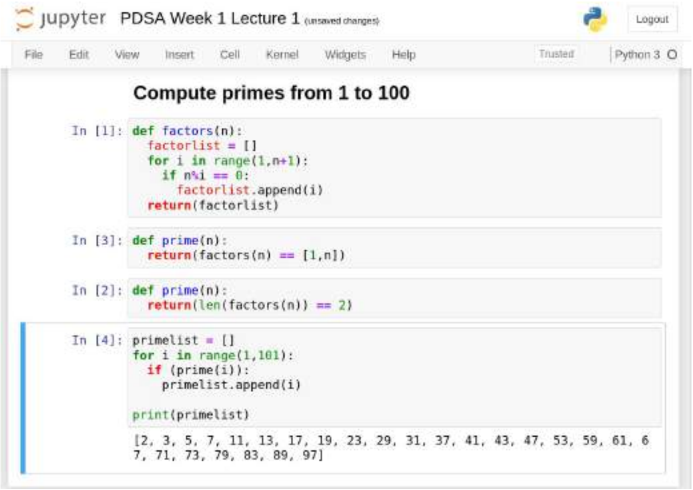
Jupyter notebook

- A sequence of cells
  - Like a one dimensional spreadsheet
- Cells hold code or text
  - Markdown notation for formatting
  - `https://www.markdownguide.org/`



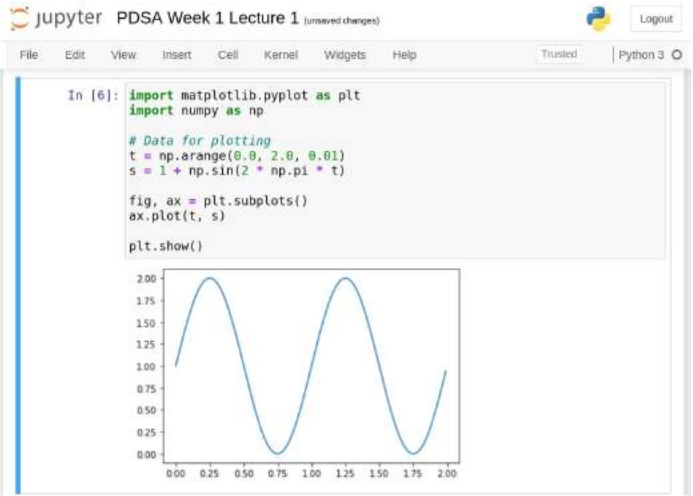
Jupyter notebook

- A sequence of cells
  - Like a one dimensional spreadsheet
- Cells hold code or text
  - Markdown notation for formatting
  - `https://www.markdownguide.org/`
- Edit and re-run individual cells to update environment



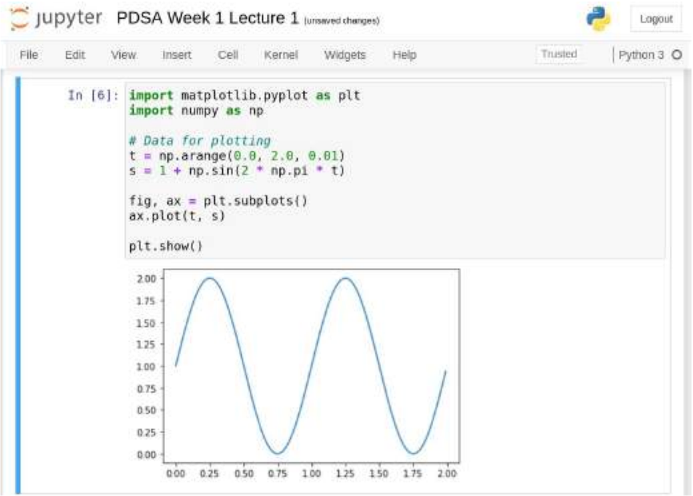
Jupyter notebook . . .

- Supports different kernels
  - Julia, Python, R
  - We will use it only for Python



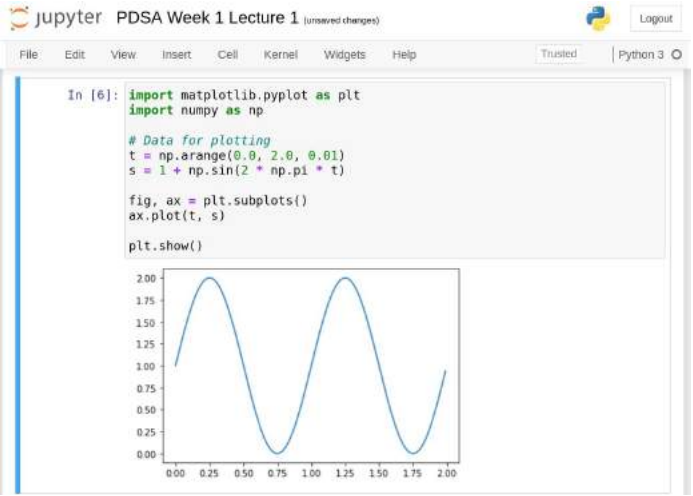
Jupyter notebook . . .

- Supports different kernels
  - Julia, Python, R
  - We will use it only for Python
- Widely used to document and disseminate ML projects
  - Solutions to problems posed on platforms like Kaggle `https://www.kaggle.org`



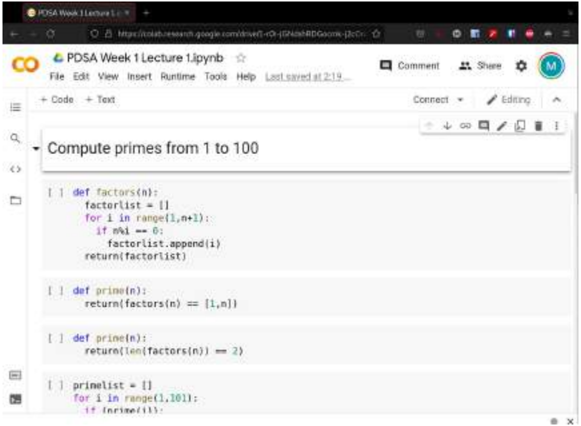
Jupyter notebook . . .

- Supports different kernels
  - Julia, Python, R
  - We will use it only for Python
- Widely used to document and disseminate ML projects
  - Solutions to problems posed on platforms like Kaggle `https://www.kaggle.org`
- ACM Software Systems Award 2017

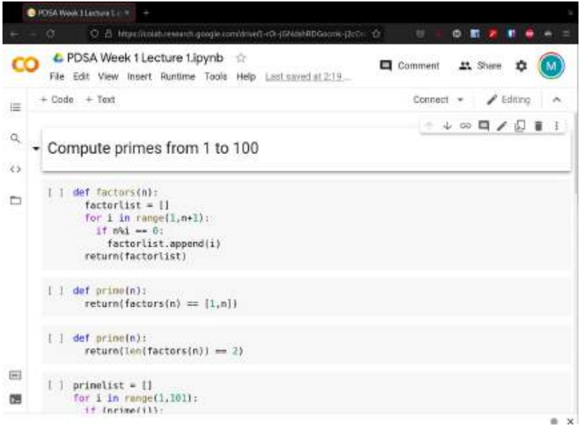


Google Colab

- Google Colaboratory (Colab)
  - `colab.research.google.com`
  - Free to use

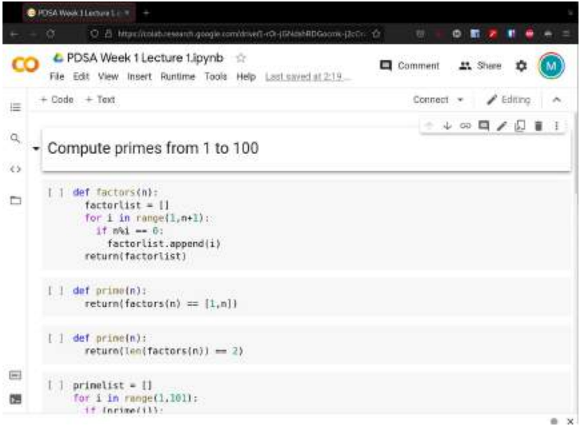


- Google Colaboratory (Colab)
  - `colab.research.google.com`
  - Free to use
- Customized Jupyter notebook



Google Colab

- Google Colaboratory (Colab)
  - `colab.research.google.com`
  - Free to use
- Customized Jupyter notebook
- All standard packages required for ML are preloaded
  - `scikit-learn`, `tensorflow`
  - Access to GPU hardware



Summary

- Jupyter notebook is a convenient interface to develop Python code

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown
- Preserve outputs when exporting

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown
- Preserve outputs when exporting
- Useful for collaboration, sharing

Summary

- Jupyter notebook is a convenient interface to develop Python code
- Incrementally update and run
- Embed documentation using Markdown
- Preserve outputs when exporting
- Useful for collaboration, sharing
- Google Colab — free to use version configured for ML

Python Recap – I

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

Computing gcd

- $\gcd(m, n)$ — greatest common divisor
  - Largest k that divides both m and n
  - $\gcd(8, 12) = 4$
  - $\gcd(18, 25) = 1$
  - Also hcf — highest common factor

Computing gcd

- $\gcd(m, n)$ — greatest common divisor
  - Largest k that divides both m and n
  - $\gcd(8, 12) = 4$
  - $\gcd(18, 25) = 1$
  - Also hcf — highest common factor
- $\gcd(m, n)$ always exsits
  - 1 divides both m and n

Computing gcd

- $\text{gcd}(m, n)$ — greatest common divisor
  - Largest k that divides both m and n
  - $\text{gcd}(8, 12) = 4$
  - $\text{gcd}(18, 25) = 1$
  - Also hcf — highest common factor
- $\text{gcd}(m, n)$ always exsits
  - 1 divides both m and n
- Computing $\text{gcd}(m, n)$
  - $\text{gcd}(m, n) \leq \text{min}(m, n)$
  - Compute list of common factors from 1 to $\text{min}(m, n)$
  - Return the last such common factor

```python
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
  - Variables (names) derive their type from the value they hold

```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
  - Variables (names) derive their type from the value they hold
- Control flow
  - Conditionals (`if`)
  - Loops (`for`)

```
def gcd(m,n):
  cf = []    # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
  - Variables (names) derive their type from the value they hold
- Control flow
  - Conditionals (`if`)
  - Loops (`for`)
- `range(i,j)` runs from `i` to `j-1`

```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Points to note

- Need to initialize `cf` for `cf.append()` to work
  - Variables (names) derive their type from the value they hold
- Control flow
  - Conditionals (`if`)
  - Loops (`for`)
- `range(i,j)` runs from `i` to `j-1`
- List indices run from `0` to `len(l) - 1` and backwards from `-1` to `-len(l)`

```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important

```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)

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

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)
- Recall that `1` is always a common factor
  - No need to initialize `mrcf`

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

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)
- Recall that `1` is always a common factor
  - No need to initialize `mrcf`

Efficiency

- Both versions of `gcd` take time proportional to $\min(m, n)$

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

```
def gcd(m,n):
  cf = []    # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Computing gcd

Eliminate the list

- Only the last value of `cf` is important
- Keep track of most recent common factor (`mrcf`)
- Recall that `1` is always a common factor
  - No need to initialize `mrcf`

Efficiency

- Both versions of `gcd` take time proportional to $\min(m, n)$
- Can we do better?

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

```python
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

Python Recap – II

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

Checking primality

- A prime number n has exactly two factors, 1 and n
  - Note that 1 is not a prime

Checking primality

- A prime number n has exactly two factors, 1 and n
  - Note that 1 is not a prime
- Compute the list of factors of `n`

```
def factors(n):
  fl = []   # factor list
  for i in range(1,n+1):
    if (n%i) == 0:
      fl.append(i)
  return(fl)
```

Checking primality

- A prime number n has exactly two factors, 1 and n
  - Note that 1 is not a prime
- Compute the list of factors of `n`
- `n` is a prime if the list of factors is precisely `[1,n]`

```
def factors(n):
  fl = []   # factor list
  for i in range(1,n+1):
    if (n%i) == 0:
      fl.append(i)
  return(fl)

def prime(n):
  return(factors(n) == [1,n])
```

Counting primes

- List all primes upto m

```
def primesupto(m):
  pl = []   # prime list
  for i in range(1,m+1):
    if prime(i):
      pl.append(i)
  return(pl)
```

Counting primes

- List all primes upto *m*
- List the first *m* primes
  - Multiple simultaneous assignment

```
def primesupto(m):
  pl = []   # prime list
  for i in range(1,m+1):
    if prime(i):
      pl.append(i)
  return(pl)

def firstprimes(m):
  (count,i,pl) = (0,1,[])
  while (count < m):
    if prime(i):
      (count,pl) = (count+1,pl+[i])
    i = i+1
  return(pl)
```

Counting primes

- List all primes upto m
- List the first m primes
  - Multiple simultaneous assignment
- `for` vs `while`
  - Is the number of iterations known in advance?
  - Ensure progress to guarantee termination of `while`

```
def primesupto(m):
  pl = []    # prime list
  for i in range(1,m+1):
    if prime(i):
      pl.append(i)
  return(pl)

def firstprimes(m):
  (count,i,pl) = (0,1,[])
  while (count < m):
    if prime(i):
      (count,pl) = (count+1,pl+[i])
    i = i+1
  return(pl)
```

Computing primes

- Directly check if n has a factor between 2 and $n-1$

```
def prime(n):
  result = True
  for i in range(2,n):
    if (n%i) == 0:
      result = False
  return(result)
```

Computing primes

- Directly check if n has a factor between 2 and $n-1$
- Terminate check after we find first factor
  - Breaking out of a loop

```
def prime(n):
  result = True
  for i in range(2,n):
    if (n%i) == 0:
      result = False
  return(result)
```

```
def prime(n):
  result = True
  for i in range(2,n):
    if (n%i) == 0:
      result = False
      break   # Abort loop
  return(result)
```

Computing primes

- Directly check if n has a factor between 2 and $n-1$
- Terminate check after we find first factor
  - Breaking out of a loop
- Alternatively, use `while`

```
def prime(n):
  result = True
  for i in range(2,n):
    if (n%i) == 0:
      result = False
      break    # Abort loop
  return(result)
```

```
def prime(n):
  (result,i) = (True,2)
  while (result and (i < n)):
    if (n%i) == 0:
      result = False
    i = i+1
  return(result)
```

Computing primes

- Directly check if n has a factor between 2 and $n-1$
- Terminate check after we find first factor
  - Breaking out of a loop
- Alternatively, use `while`
- Speeding things up slightly
  - Factors occur in pairs
  - Sufficient to check factors upto $\sqrt{n}$
  - If n is prime, scan $2, \ldots, \sqrt{n}$ instead of $2, \ldots, n-1$

```
import math
def prime(n):
  (result,i) = (True,2)
  while (result and (i < math.sqrt(n))
    if (n%i) == 0:
      result = False
    i = i+1
  return(result)
```

Properties of primes

- There are infinitely many primes

Properties of primes

- There are infinitely many primes
- How are they distributed?

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: p, $p+2$

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: p, $p+2$
- Twin prime conjecture
  There are infinitely many twin primes

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: p, $p+2$
- Twin prime conjecture
  There are infinitely many twin primes
- Compute the differences between primes

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: p, $p+2$
- Twin prime conjecture
  There are infinitely many twin primes
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: p, $p + 2$
- Twin prime conjecture
  There are infinitely many twin primes
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

```
def primediffs(n):
  lastprime = 2
  pd = {}  # Dictionary for
           # prime diferences
  for i in range(3,n+1):
    if prime(i):
      d = i - lastprime
      lastprime = i
      if d in pd.keys():
        pd[d] = pd[d] + 1
      else:
        pd[d] = 1
  return(pd)
```

Python Recap – III

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

Computing gcd

- Both versions of `gcd` take time proportional to $\min(m, n)$
- Can we do better?

```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

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

Computing gcd

- Both versions of `gcd` take time proportional to $\min(m, n)$
- Can we do better?
- Suppose d divides m and n
  - $m = ad$, $n = bd$
  - $m - n = (a - b)d$
  - d also divides $m - n$

```
def gcd(m,n):
  cf = []   # List of common factors
  for i in range(1,min(m,n)+1):
    if (m%i) == 0 and (n%i) == 0:
      cf.append(i)
  return(cf[-1])
```

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

Computing gcd

- Both versions of gcd take time proportional to $\min(m, n)$
- Can we do better?
- Suppose d divides m and n
  - $m = ad$, $n = bd$
  - $m - n = (a - b)d$
  - d also divides $m - n$
- Recursively defined function
  - Base case: n divides m, answer is n
  - Otherwise, reduce $gcd(m, n)$ to $gcd(n, m - n)$

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider `gcd(2,9999)`
  - $\rightarrow$ `gcd(2,9997)`
  - $\rightarrow$ `gcd(2,9995)`
  - ...
  - $\rightarrow$ `gcd(2,3)`
  - $\rightarrow$ `gcd(2,1)`
  - $\rightarrow$ 1

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider `gcd(2,9999)`
  - $\rightarrow$ `gcd(2,9997)`
  - $\rightarrow$ `gcd(2,9995)`
  - ...
  - $\rightarrow$ `gcd(2,3)`
  - $\rightarrow$ `gcd(2,1)`
  - $\rightarrow$ 1
- Approximately 5000 steps

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a-b))
```

Computing gcd

- Unfortunately, this takes time proportional to $\max(m, n)$
- Consider `gcd(2,9999)`
  - $\rightarrow$ `gcd(2,9997)`
  - $\rightarrow$ `gcd(2,9995)`
  - ...
  - $\rightarrow$ `gcd(2,3)`
  - $\rightarrow$ `gcd(2,1)`
  - $\rightarrow$ 1
- Approximately 5000 steps
- Can we do better?

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a-b))
```

Euclid's algorithm

- Suppose n does not divide m

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
  - If n divides m, $\text{gcd}(m, n) = n$
  - Otherwise, compute $\text{gcd}(n, m \bmod n)$

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a%b))
```

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
  - If n divides m, $\text{gcd}(m, n) = n$
  - Otherwise, compute $\text{gcd}(n, m \bmod n)$

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a%b))
```

- Can show that this takes time proportional to number of digits in $\max(m, n)$

Euclid's algorithm

- Suppose n does not divide m
- Then $m = qn + r$
- Suppose d divides both m and n
- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's algorithm
  - If n divides m, $\gcd(m, n) = n$
  - Otherwise, compute $\gcd(n, m \bmod n)$

```
def gcd(m,n):
  (a,b) = (max(m,n), min(m,n))
  if a%b == 0:
    return(b)
  else
    return(gcd(b,a%b))
```

- Can show that this takes time proportional to number of digits in $\max(m, n)$
- One of the first non-trivial algorithms

Exception handling

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

When things go wrong

- Our code could generate many types of errors
  - `y = x/z`, but `z` has value `0`
  - `y = int(s)`, but string `s` does not represent a valid integer
  - `y = 5*x`, but `x` does not have a value
  - `y = l[i]`, but `i` is not a valid index for list `l`
  - Try to read from a file, but the file does not exist
  - Try to write to a file, but the disk is full

When things go wrong

- Our code could generate many types of errors
  - `y = x/z`, but `z` has value `0`
  - `y = int(s)`, but string `s` does not represent a valid integer
  - `y = 5*x`, but `x` does not have a value
  - `y = l[i]`, but `i` is not a valid index for list `l`
  - Try to read from a file, but the file does not exist
  - Try to write to a file, but the disk is full
- Recovering gracefully
  - Try to anticipate errors
  - Provide a contingency plan
  - Exception handling

Types of errors

- Python flags the type of each error

Types of errors

- Python flags the type of each error
- Most common error is a syntax error
  - `SyntaxError: invalid syntax`
  - Not much you can do!

Types of errors

- Python flags the type of each error
- Most common error is a syntax error
  - `SyntaxError: invalid syntax`
  - Not much you can do!
- We are interested in errors when the code is running
  - Name used before value is defined
    `NameError: name 'x' is not defined`
  - Division by zero in arithmetic expression
    `ZeroDivisionError: division by zero`
  - Invalid list index
    `IndexError: list assignment index out of range`

Terminology

- Raise an exception
  - Run time error $\rightarrow$ signal error type, with diagnostic information
    `NameError: name 'x' is not defined`
- Handle an exception
  - Anticipate and take corrective action based on error type
- Unhandled exception aborts execution

Terminology

- Raise an exception
  - Run time error $\rightarrow$ signal error type, with diagnostic information `NameError: name 'x' is not defined`
- Handle an exception
  - Anticipate and take corrective action based on error type
- Unhandled exception aborts execution

Handling exceptions

```
try:
   ... ← Code where error may occur
   ...

except IndexError:
   ... ← Handle IndexError

except (NameError,KeyError):
   ... ← Handle multiple exception types

except:
   ... ← Handle all other exceptions

else:
   ... ← Execute if try runs without errors
```

Using exceptions "positively"

- Collect scores in dictionary

```
scores = {"Shefali":[3,22],
          "Harmanpreet":[200,3]}
```

- Update the dictionary
- Batter `b` already exists, append to list

```
scores[b].append(s)
```

- New batter, create a fresh entry

```
scores[b] = [s]
```

Using exceptions "positively"

- Collect scores in dictionary

```
scores = {"Shefali":[3,22],
          "Harmanpreet":[200,3]}
```

- Update the dictionary
- Batter `b` already exists, append to list

```
scores[b].append(s)
```

- New batter, create a fresh entry

```
scores[b] = [s]
```

Traditional approach

```
if b in scores.keys():
  scores[b].append(s)
else:
  scores[b] = [s]
```

Using exceptions "positively"

- Collect scores in dictionary

```
scores = {"Shefali":[3,22],
          "Harmanpreet":[200,3]}
```

- Update the dictionary
- Batter `b` already exists, append to list

```
scores[b].append(s)
```

- New batter, create a fresh entry

```
scores[b] = [s]
```

Traditional approach

```
if b in scores.keys():
  scores[b].append(s)
else:
  scores[b] = [s]
```

Using exceptions

```
try:
  scores[b].append(s)
except KeyError:
  scores[b] = [s]
```

Flow of control

```
...
x = f(y,z)
```

Flow of control

```
...
x = f(y,z)
```

```
def f(a,b):
  ...
  g(a)
```

Flow of control

```
...
x = f(y,z)
```

```
def f(a,b):
  ...
  g(a)
```

```
def g(m):
  ...
  h(m)
```

Flow of control

```
...
x = f(y,z)
```

```
def f(a,b):
  ...
  g(a)
```

```
def g(m):
  ...
  h(m)
```

```
def h(s):
  ...
  h(s)
```

Flow of control

```
...
x = f(y,z)
```

```
def f(a,b):
  ...
  g(a)
```

```
def g(m):
  ...
  h(m)
```

```
def h(s):
  ...
  h(s)
```

`IndexError` not handled in `h()`

Flow of control

```
...
x = f(y,z)
```

```
def f(a,b):
  ...
  g(a)
```

```
def g(m):
  ...
  h(m)
```

`IndexError` inherited from `h()`

```
def h(s):
  ...
  h(s)
```

`IndexError` not handled in `h()`

Flow of control

```
...
x = f(y,z)
```

```
def f(a,b):
  ...
  g(a)
```

`IndexError` inherited from `g()`

```
def g(m):
  ...
  h(m)
```

`IndexError` inherited from `h()`

Not handled?

```
def h(s):
  ...
  h(s)
```

`IndexError` not handled in `h()`

Flow of control

```
...
x = f(y,z)
```

`IndexError` inherited from `f()`

```
def f(a,b):
  ...
  g(a)
```

`IndexError` inherited from `g()`
Not handled?

```
def g(m):
  ...
  h(m)
```

`IndexError` inherited from `h()`
Not handled?

```
def h(s):
  ...
  h(s)
```

`IndexError` not handled in `h()`

Flow of control

```
...
x = f(y,z)
```

`IndexError` inherited from `f()`

Not handled?

Abort!

```
def f(a,b):
  ...
  g(a)
```

`IndexError` inherited from `g()`

Not handled?

```
def g(m):
  ...
  h(m)
```

`IndexError` inherited from `h()`

Not handled?

```
def h(s):
  ...
  h(s)
```

`IndexError` not handled in `h()`

Classes and objects

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, stack: last-in, first-out, `push()`, `pop()`

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, stack: last-in, first-out, `push()`, `pop()`
- Separate the (private) implementation from the (public) specification

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, stack: last-in, first-out, `push()`, `pop()`
- Separate the (private) implementation from the (public) specification
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data

Classes and objects

- Abstract datatype
  - Stores some information
  - Designated functions to manipulate the information
  - For instance, stack: last-in, first-out, `push()`, `pop()`
- Separate the (private) implementation from the (public) specification
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template

Example: 2D points

- A point has coordinates (x, y)
  - `__init__()` initializes internal values `x`, `y`
  - First parameter is always `self`
  - Here, by default a point is at $(0, 0)$

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

Example: 2D points

- A point has coordinates (x, y)
  - `__init__()` initializes internal values `x`, `y`
  - First parameter is always `self`
  - Here, by default a point is at $(0, 0)$
- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    self.x += deltax
    self.y += deltay
```

Example: 2D points

- A point has coordinates (x, y)
  - `__init__()` initializes internal values `x`, `y`
  - First parameter is always `self`
  - Here, by default a point is at $(0, 0)$
- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    self.x += deltax
    self.y += deltay

  def odistance(self):
    import math
    d = math.sqrt(self.x*self.x +
                  self.y*self.y)
    return(d)
```

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)

  def odistance(self):
    return(self.r)
```

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r\cos\theta$, $y = r\sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$

```
def translate(self,deltax,deltay):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += deltax
  y += deltay
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Polar coordinates

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r \cos \theta$, $y = r \sin \theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not be aware whether representation is (x, y) or (r, θ)

```
def translate(self,deltax,deltay):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += deltax
  y += deltay
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Special functions

- `__init__()` — constructor

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
  - `str(o) == o.__str()__`
  - Implicitly invoked by `print()`

```
class Point:

  ...

  def __str__(self):
    return(
      '('+str(self.x)+','
         +str(self.y)+')'
    )
```

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
  - `str(o) == o.__str()__`
  - Implicitly invoked by `print()`
- `__add__()`
  - Implicitly invoked by `+`

```
class Point:

  ...

  def __str__(self):
    return(
      '('+str(self.x)+','
        +str(self.y)+')'
    )

  def __add__(self,p):
    return(Point(self.x + p.x,
                 self.y + p.y))
```

Special functions

- `__init__()` — constructor
- `__str__()` — convert object to string
  - `str(o) == o.__str()__`
  - Implicitly invoked by `print()`
- `__add__()`
  - Implicitly invoked by `+`
- Similarly
  - `__mult__()` invoked by `*`
  - `__lt__()` invoked by `<`
  - `__ge__()` invoked by `>=`
  - ...

```
class Point:

  ...

  def __str__(self):
    return(
      '('+str(self.x)+','
         +str(self.y)+')'
    )

  def __add__(self,p):
    return(Point(self.x + p.x,
                 self.y + p.y))
```

Timing our code

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

Timing our code

- How long does our code take to execute?
    - Depends from one language to another

Timing our code

- How long does our code take to execute?
  - Depends from one language to another
- Python has a library `time` with various useful functions

Timing our code

- How long does our code take to execute?
  - Depends from one language to another
- Python has a library `time` with various useful functions
- `perf_time()` is a performance counter
  - Absolute value of `perf_time()` is not meaningful
  - Compare two consecutive readings to get an interval
  - Default unit is seconds

Timing our code

- How long does our code take to execute?
  - Depends from one language to another
- Python has a library `time` with various useful functions
- `perf_time()` is a performance counter
  - Absolute value of `perf_time()` is not meaningful
  - Compare two consecutive readings to get an interval
  - Default unit is seconds

```
import time

start = time.perf_counter()
...
# Execute some code
...
end = time.perf_counter()

elapsed = end - start
```

A timer object

- Create a timer class

```
import time
class Timer:
```

A timer object

- Create a timer class
- Two internal values
  - _start_time
  - _elapsed_time

```
import time
class Timer:

  def __init__(self):
    self._start_time = 0
    self._elapsed_time = 0
```

A timer object

- Create a timer class
- Two internal values
  - `_start_time`
  - `_elapsed_time`
- `start` starts the timer

```
import time
class Timer:

  def __init__(self):
    self._start_time = 0
    self._elapsed_time = 0

  def start(self):
    self._start_time = time.perf_counter()
```

A timer object

- Create a timer class
- Two internal values
  - `_start_time`
  - `_elapsed_time`
- `start` starts the timer
- `stop` records the elapsed time

```
import time
class Timer:

  def __init__(self):
    self._start_time = 0
    self._elapsed_time = 0

  def start(self):
    self._start_time = time.perf_counter()

  def stop(self):
    self._elapsed_time =
       time.perf_counter() - self._start_time

  def elapsed(self):
    return(self._elapsed_time)
```

A timer object

- Create a timer class
- Two internal values
  - `_start_time`
  - `_elapsed_time`
- `start` starts the timer
- `stop` records the elapsed time
- More sophisticated version in the actual code

```
import time
class Timer:

  def __init__(self):
    self._start_time = 0
    self._elapsed_time = 0

  def start(self):
    self._start_time = time.perf_counter()

  def stop(self):
    self._elapsed_time =
       time.perf_counter() - self._start_time

  def elapsed(self):
    return(self._elapsed_time)
```

A timer object

- Create a timer class
- Two internal values
  - `_start_time`
  - `_elapsed_time`
- `start` starts the timer
- `stop` records the elapsed time
- More sophisticated version in the actual code
- Python executes 10^7 operations per second

```
import time
class Timer:

  def __init__(self):
    self._start_time = 0
    self._elapsed_time = 0

  def start(self):
    self._start_time = time.perf_counter()

  def stop(self):
    self._elapsed_time =
        time.perf_counter() - self._start_time

  def elapsed(self):
    return(self._elapsed_time)
```

Why Efficiency Matters

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 1

A real world problem

- Every SIM card needs to be linked to an Aadhaar card

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop
- How long will this take?
  - M SIM cards, N Aadhaar cards
  - Nested loops iterate $M \cdot N$ times

```
for each SIM card S:
  for each Aadhaar number A:
    check if Aadhaar details of S
    match A
```

A real world problem

- Every SIM card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop
- How long will this take?
  - M SIM cards, N Aadhaar cards
  - Nested loops iterate $M \cdot N$ times
- What are M and N
  - Almost everyone in India has an Aadhaar card: $N > 10^9$
  - Number of SIM cards registered is similar: $M > 10^9$

```
for each SIM card S:
  for each Aadhaar number A:
    check if Aadhaar details of S
    match A
```

A real world problem

- Assume $M = N = 10^9$

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
  - $10^{11}/60 \approx 1.67 \times 10^9$ minutes

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
  - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
  - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours

```
for each SIM card S:
  for each Aadhaar number A:
    check if Aadhaar details of S
    match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
  - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
  - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours
  - $(2.8 \times 10^7)/24 \approx 1.17 \times 10^6$ days

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
  - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
  - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours
  - $(2.8 \times 10^7)/24 \approx 1.17 \times 10^6$ days
  - $(1.17 \times 10^6)/365 \approx 3200$ years!

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

A real world problem

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
  - $10^{11}/60 \approx 1.67 \times 10^9$ minutes
  - $(1.67 \times 10^9)/60 \approx 2.8 \times 10^7$ hours
  - $(2.8 \times 10^7)/24 \approx 1.17 \times 10^6$ days
  - $(1.17 \times 10^6)/365 \approx 3200$ years!
- How can we fix this?

```
for each SIM card S:
  for each Aadhaar number A:
     check if Aadhaar details of S
     match A
```

Guess my birthday

- You propose a date

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities
- Query midpoint — halves the interval

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier
  - April 11? Later

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?
- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier
  - April 11? Later
  - April 16? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier
  - April 11? Later
  - April 16? Earlier
  - April 13? Earlier

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier
  - April 11? Later
  - April 16? Earlier
  - April 13? Earlier
  - April 12? Yes

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier
  - April 11? Later
  - April 16? Earlier
  - April 13? Earlier
  - April 12? Yes
- Interval shrinks from $365 \rightarrow 182 \rightarrow 91 \rightarrow 45 \rightarrow 22 \rightarrow 11 \rightarrow 5 \rightarrow 2 \rightarrow 1$

Guess my birthday

- You propose a date
- I answer, *Yes*, *Earlier*, *Later*
- Suppose my birthday is 12 April
- A possible sequence of questions
  - September 12? Earlier
  - February 23? Later
  - July 2? Earlier
  - ...
- What is the best strategy?

- Interval of possibilities
- Query midpoint — halves the interval
  - June 30? Earlier
  - March 31? Later
  - May 15? Earlier
  - April 22? Earlier
  - April 11? Later
  - April 16? Earlier
  - April 13? Earlier
  - April 12? Yes
- Interval shrinks from $365 \rightarrow 182 \rightarrow 91 \rightarrow 45 \rightarrow 22 \rightarrow 11 \rightarrow 5 \rightarrow 2 \rightarrow 1$
- Under 10 questions

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

```
for each SIM card S:
   probe sorted Aadhaar list to
   check Aadhaar details of S
```

- 3000 seconds, or 50 minutes

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

```
for each SIM card S:
   probe sorted Aadhaar list to
   check Aadhaar details of S
```

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

```
for each SIM card S:
  probe sorted Aadhaar list to
  check Aadhaar details of S
```

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards
- Arranging the data results in a much more efficient solution

A real world problem

- Assume Aadhaar details are sorted by Aadhaar number
- Use the halving strategy to check each SIM card
- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$
- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

```
for each SIM card S:
    probe sorted Aadhaar list to
    check Aadhaar details of S
```

- 3000 seconds, or 50 minutes
- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards
- Arranging the data results in a much more efficient solution
- Both algorithms and data structures matter