Week 1

PYTHON RECAP

1. Computing GCD

- `gcd(m,n)` <= `min(m,n)`
- Compute the list of common factors from 1 to `min(m,n)`
- Return the last such common factor

```
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]
```

Points to note:

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

Modifications:

- Eliminate the list
  - Only the last value of cf is important.
- Keep track of most recent commaon factor (`mrcf`)
  - 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
```

Efficiency:

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

2. Check primality

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

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

- `n` is a prime if the list of factors is precisely `[1,n]`

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

Another method to check primality:

- 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
      break # Abort loop
  return result
```

Alternate Method 1:

- Use `while` loop
- Avoid using `break` to avoid confusion.

```
def prime(n):
  (result,i) = (True,2)
```

```
  while (result and (i < n)):
    if (n%i) == 0:
      result = False
    i = i + 1
  return result
```

Alternate Method 2: Speeding things up slightly

- Factors occer in pairs
- Sufficient to check factors upto $\sqrt{n}$
- If *n* is prime, scan 2, ..., $\sqrt{n}$ instead of 2, ..., *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
```

3. Counting primes

[ ] ↳ *15 cells hidden*

EXCEPTION HANDLING

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 doesn't exist.
- Try to write to file, but disk is full.

Recover gracefully

- Try to anticipate errors.
- Provide a contingency plan.
- **Exception handling**

Types of Errors:

- Python flags the type of each error.

- Most common error is a syntax error:
  - `SyntaxError: invalid syntax`
  - Not much we can do.
- Errors when code is running:
  1. Name used before value is defined
     - `NameError: name 'x' is not defined`
  2. Division by zero in arithmetic expression
     - `ZeroDivisionError: division by zero`
  3. Invalid list index
     - `IndexError: list assignment index out of range`

Terminology:

- Raise an exception
  - Run time error -> signal error type, with diagnostic information.
- 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:

Traditional approach:

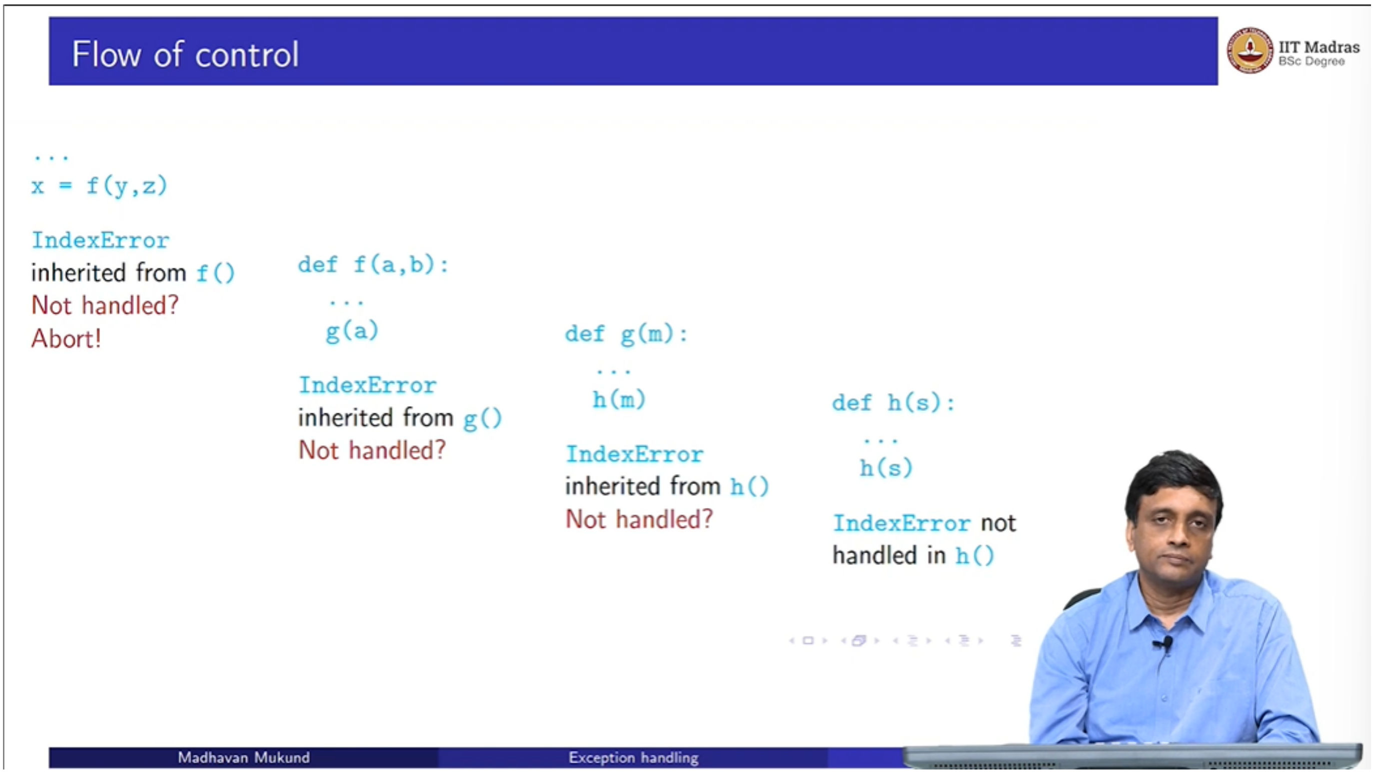
- Collect scores in dictionary
  `scores = {"Shefali": [3,22], "Harmanpreet": [200,3]}`
- Update the dictionary

- Batter *b* already exists, append to list
  `scores[b].append(s)`
- New batter, create a fresh entry
  `scores[b] = [s]`

Using exceptions:

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

Flow of Control:



CLASSES AND OBJECTS

Abstract datatype

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

Class:

- Template for a datatype
- 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
```

Translation: Shift a point by (Δx, Δy)

- `(x,y)` -> (x + Δx, y + Δy)

```
def translate(self, deltax, deltay): # part of the class defn.
  self.x += deltax
  self.y += deltay
```

Distance from origin:

- *d* = $\sqrt{x^2 + y^2}$

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

```
p = Point(3,4)
q = point(7,10)
# print(p + q) will be an error
p.odistance(), q.odistance()
```

Polar coordinates:
(r, θ) instead of `(x,y)`

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

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

```
def odistance(self): # Distance from origin is just r
  return self.r
```

Translation with polar coordinates:

- Convert (r, θ) to `(x,y)`
- $x = r \cos\theta, y = r \sin\theta$
- Recompute r , θ from (x + Δx, y + Δy)

```
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)
```

```
p = Point(3,4)
q = Point(7,10)
p.odistance(), q.odistance()
p.r, p.theta
```

Interface has not changed!

- User need not be aware whether representation is `(x,y)` or (r, θ)

Special functions:

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

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

- `__add__()`
    - Implicitly invoked by `+`

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

Similarly,

- `__mult__()` invoked by `*`
- `__lt__()` invoked by `<`
- `__ge__()` invoked by `>=`
- ...

TIMING OUR CODE:

- How long our code takes to execute depends on the language we use.
- `time` library with various functions.

- `perf.time()` is a performance counter
    - Absolute value of `perf.time()` is not meaningful
    - Compaer 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
- 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
```

WHY EFFICIENCY MATTERS?

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
```

- How long will this take?
  - *M* SIM cards, *N* Aadhaar cards
  - Nested loops iterate *M* . *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
- Assume *M* = *N* = 10^9
- Nested loops execute 10^{18} times.
- Python can perform 10^7 operations in a second (calculated from Timer class)
- This will take at least 10^{11} seconds.
  - 10^{11}/60 $\approx$ 1.67 x 10^9 minutes
  - (1.67 x 10^9)/60 $\approx$ 2.8 x 10^7 hours
  - (2.8 x 10^7)/60 $\approx$ 1.17 x 10^6 days
  - (1.17 x 10^6)/365 $\approx$ 3200 years!

Halving strategy:

- Interval of possibilities.
- Query midpoint - halves the intervals
- Keeps checking the mid points and halving the intervals till it finds a match
- Interval shrinks, time taken less!

How to solve this?

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

  ```
  for each SIM card S:
    probe sorted Aadhaar list to
    check Aadhaar details of S
  ```
- 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 $\approx 10^9$ x 30 seconds $\approx$ 50 minutes!
- Of course to achieve this, we have to first sort the Aadhaar cards.
- **Conclusion:** *Arranging the data results in a much more efficient solution. Both algorithms and data structures matter.*