INDIAN INSTITUTE OF TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Summary of concepts introduced in the course

What is computational thinking?

- Expressing problem solutions as a sequence of steps for communication to a computer
- Finding common patterns between solutions, apply these patterns to solve new problems

What is computational thinking?

- Expressing problem solutions as a sequence of steps for communication to a computer
- Finding common patterns between solutions, apply these patterns to solve new problems
- Data science problems are usually posed on a dataset
    - which can be obtained from real life artefacts - time tables, shopping bills, transaction logs
    - ... or may be available in a digital format - typically in the form of tables

What is computational thinking?

- Expressing problem solutions as a sequence of steps for communication to a computer
- Finding common patterns between solutions, apply these patterns to solve new problems
- Data science problems are usually posed on a dataset
  - which can be obtained from real life artefacts - time tables, shopping bills, transaction logs
  - ... or may be available in a digital format - typically in the form of tables
- Computational thinking in datascience involves finding patterns in methods used to process these datasets
- Through this course, several concepts and methods were introduced for doing this
  - Typically involves first scanning the dataset to collect relevant information
  - Then processing this information to find relationships between data elements
  - Finally organising the relationships in a form that allows questions to be answered easily

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
    - Iterator needs to be initialised
    - The steps that need to be repeated need to be made precise
    - We should know when and how to exit from an iteration, and where to go after the exit

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit
- **Variables** are storage units used to keep track of intermediate values during the iteration

Iterators and Variables

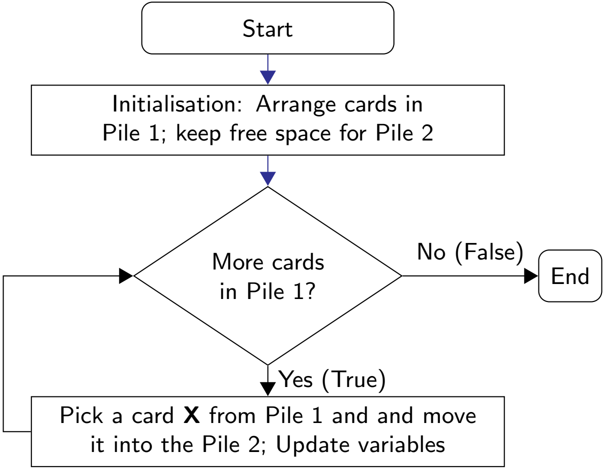
- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit
- **Variables** are storage units used to keep track of intermediate values during the iteration
- Initialisation and updates of variables are done through **assignment statements**

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit
- **Variables** are storage units used to keep track of intermediate values during the iteration
- Initialisation and updates of variables are done through **assignment statements**
- Variables which assemble a value or a collection are called **accumulators**

Pseudocode and flowchart for processing a dataset

```
Initialise variables
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  Update values of variables
}
```


Start
Initialisation: Arrange cards in Pile 1; keep free space for Pile 2
More cards in Pile 1?
No (False)
End
Yes (True)
Pick a card **X** from Pile 1 and and move it into the Pile 2; Update variables

The set of items need to have well defined values

- Variables can be of different **datatypes**
- Basic data types: **boolean, integer, character** and **string**
- **Subtypes** put more constraints on the values and operations allowed
- Lists and Records are two ways of creating bigger bundles of data
- In a **list** all data items typically have the same datatype
- Whereas, a **record** has multiple named fields, each can be of a different datatype
- A **Dictionary** is like a record to which we can add new fields

Iteration with Filtering

- **Filtering** allows us to select the relevant data elements for processing and ignore the rest
- Requires complex boolean conditions to be defined
  - We can make compound conditions using boolean connectives - and, or, not
  - ... or we can do condition checking in sequence
  - ... or we can make nested condition checks

Iteration with Filtering

- **Filtering** allows us to select the relevant data elements for processing and ignore the rest
- Requires complex boolean conditions to be defined
  - We can make compound conditions using boolean connectives - and, or, not
  - ... or we can do condition checking in sequence
  - ... or we can make nested condition checks
- The conditions can work on the (fixed) data elements or on variables:
  - Compare the item values with a constant - example count, sum. The filtering condition does not change after each iteration step
  - compare item values with a variable - example max. The filtering condition changes after an iteration step

Iteration with Filtering

- **Filtering** allows us to select the relevant data elements for processing and ignore the rest
- Requires complex boolean conditions to be defined
  - We can make compound conditions using boolean connectives - and, or, not
  - ... or we can do condition checking in sequence
  - ... or we can make nested condition checks
- The conditions can work on the (fixed) data elements or on variables:
  - Compare the item values with a constant - example count, sum. The filtering condition does not change after each iteration step
  - compare item values with a variable - example max. The filtering condition changes after an iteration step
- Using filtering with accumulation, we can assemble a lot of intermediate information about the dataset

Procedures and parameters

- Sometimes we have to write the same piece of code again and again with small differences
- A piece of pseudocode can be converted into a **procedure** by separating it out from the rest of the code
- Some variables (or constants) used in this piece of code can be replaced by a **parameter**, so that the same procedure can be called with different parameter values to work on different data elements

Procedures and parameters

- Sometimes we have to write the same piece of code again and again with small differences
- A piece of pseudocode can be converted into a **procedure** by separating it out from the rest of the code
- Some variables (or constants) used in this piece of code can be replaced by a **parameter**, so that the same procedure can be called with different parameter values to work on different data elements
- A procedure can return a value, and the return value can be assigned to a variable. This makes the code much more compact and easy to read and manage

Procedures and parameters

- Sometimes we have to write the same piece of code again and again with small differences
- A piece of pseudocode can be converted into a **procedure** by separating it out from the rest of the code
- Some variables (or constants) used in this piece of code can be replaced by a **parameter**, so that the same procedure can be called with different parameter values to work on different data elements
- A procedure can return a value, and the return value can be assigned to a variable. This makes the code much more compact and easy to read and manage
- The procedure can have **side-effects**
    - it can change the value of variables that are passed as parameters
    - or those that are made accessible to the procedure, such as the data set elements or lists and dictionaries created from them
    - Procedures with side-effects need to be used carefully

Multiple iterations

- Two iterations can be carried out in sequence or nested

Multiple iterations

- Two iterations can be carried out in sequence or nested
- In a sequential iteration, we make multiple passes through the data, using the result of one pass during the next pass.
    - first pass could collect some intermediate information, and the second pass can filter elements using this information
    - It establishes a relation between elements with the aggregate
    - e.g. find all the below average students

Multiple iterations

- Two iterations can be carried out in sequence or nested
- In a sequential iteration, we make multiple passes through the data, using the result of one pass during the next pass.
  - first pass could collect some intermediate information, and the second pass can filter elements using this information
  - It establishes a relation between elements with the aggregate
  - e.g. find all the below average students
- Nested iterations are used when we want to create a relationship between pairs of data elements
  - Nested iterations are costly in terms of number of computations required
  - We could reduce the number of comparisons by using **binning** wherever possible
  - The relationships produced through nested iterations can be stored using lists, dictionaries (or **graphs**)

Lists

- A **list** is a sequence of values
- Write a list as `[x1,x2,...,xn]`, combine lists using `++`
  - `[x1,x2] ++ [y1,y2,y3]` ↦ `[x1,x2,y1,y2,y3]`
- Extending list `l` by an item `x`
  - `l = l ++ [x]`
- `foreach` iterates through values in a list
- `length(l)` returns number of elements in `l`
- Functions to extract first and last items of a list
  - `first(l)` and `rest(l)`
  - `last(l)` and `init(l)`

Sorted lists

- **Sorting** is an important pre-processing step
- **Insertion sort** is a natural sorting algorithm
  - Repeatedly insert each item of the original list into a new sorted list
  - The list can be sorted in ascending or descending order
- Sorted lists allow simpler solutions to be found to some problems - example identify the quartiles for awarding grades

Dictionaries

- A **dictionary** stores a collection of key:value pairs
- Random access — getting the value for any key takes constant time
- Dictionary is sequence `{k1:v1, k2:v2, ..., kn:vn}`
- Usually, create an empty dictionary and add key-value pairs

```
d = {}
d[k1] = v1
d[k7] = v7
```

- Iterate through a dictionary using `keys(d)`

```
foreach k in keys(d) {
  Do something with d[k]
}
```

- `isKey(d,k)` reports whether `k` is a key in `d`

```
if isKey(d,k){
  d[k] = d[k] + v
}
else{
  d[k] = v
}
```

Graphs

- **Graphs** are a useful way to represent relationships
    - Add an edge from `i` to `j` if `i` is related to `j`

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`
- Use matrices to represent graphs
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`
- Use matrices to represent graphs
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`
- Iterate through matrix to aggregate information from the graph
- Graphs are useful to represent connectivity in a network
  - A path is a sequence of edges
  - Starting with direct edges, we can iteratively find longer and longer paths

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`
- Use matrices to represent graphs
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`
- Iterate through matrix to aggregate information from the graph
- Graphs are useful to represent connectivity in a network
  - A path is a sequence of edges
  - Starting with direct edges, we can iteratively find longer and longer paths
- We can represent extra information in a graph via **edge labels** - e.g. distance
  - Iteratively update labels - e.g. compute shortest distance path between each pair of stations

Matrices

- **Matrices** are two dimensional tables
    - Support random access to any element `m[i][j]`

Matrices

- **Matrices** are two dimensional tables
  - Support random access to any element `m[i][j]`
- We can implement matrices using nested dictionaries

Matrices

- **Matrices** are two dimensional tables
  - Support random access to any element `m[i][j]`
- We can implement matrices using nested dictionaries
- Use iterators to process matrices row-wise and column-wise
  - `foreach r in rows(mymatrix)`
  - `foreach c in columns(mymatrix)`

Recursion

- Many functions are naturally defined in an inductive manner
    - Base case and inductive step

Recursion

- Many functions are naturally defined in an inductive manner
  - Base case and inductive step
- Use **recursive procedures** to compute such functions
  - Base case: value is explicitly calculated and returned. Has to be properly defined to ensure that the recursion terminates
  - Inductive case: value requires procedure to evaluated on a smaller input
  - Suspend the current computation till the recursive computation terminates

Recursion

- Many functions are naturally defined in an inductive manner
  - Base case and inductive step
- Use **recursive procedures** to compute such functions
  - Base case: value is explicitly calculated and returned. Has to be properly defined to ensure that the recursion terminates
  - Inductive case: value requires procedure to evaluated on a smaller input
  - Suspend the current computation till the recursive computation terminates
- **Depth first search** is a systematic procedure to explore a graph
  - Recursively visit all unexplored neighbours
  - Keep track of visited vertices in a dictionary
  - Can discover properties of the graph — for instance, is it connected?

Encapsulation

- **Encapsulation** packages procedures with the data elements on which they operate
  - Fields (or procedures) can be hidden from the outside world by marking them as **private**
  - Procedures act as the interfaces to the external world

Encapsulation

- **Encapsulation** packages procedures with the data elements on which they operate
  - Fields (or procedures) can be hidden from the outside world by marking them as **private**
  - Procedures act as the interfaces to the external world
- Encapsulation provides more modularisation than procedures
  - State is retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - **Derived types** extend the functionality to specific instances
  - Disadvantage: objects need to be created and initialised at the start

Encapsulation

- **Encapsulation** packages procedures with the data elements on which they operate
  - Fields (or procedures) can be hidden from the outside world by marking them as **private**
  - Procedures act as the interfaces to the external world
- Encapsulation provides more modularisation than procedures
  - State is retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - **Derived types** extend the functionality to specific instances
  - Disadvantage: objects need to be created and initialised at the start
- "Object oriented computing" patterns can be found between different examples

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously
- This requires the **remote procedure call** to be unbundled. Two models:
  - In the **polling** model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - In the **producer-consumer** model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously
- This requires the **remote procedure call** to be unbundled. Two models:
  - In the **polling** model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - In the **producer-consumer** model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases, **race** conditions have to be handled by ensuring that access to shared objects (such as lists) are made **atomic** (i.e. non-concurrent).

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously
- This requires the **remote procedure call** to be unbundled. Two models:
  - In the **polling** model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - In the **producer-consumer** model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases, **race** conditions have to be handled by ensuring that access to shared objects (such as lists) are made **atomic** (i.e. non-concurrent).
- Concurrency can model Input/Output between users and a computer

Bottom-up computing

- In **bottom-up computing**, the code is constructed from (a sample of) the data elements
- In **classification**, a tree like structure (decision tree) is created
- **Prediction** tries to find a (linear) numerical function that connects the value to be predicted to the numerical values of the data elements that are available
- Classification and Prediction can be combined. Decision trees can use prediction functions with cutoffs, and prediction functions can be made specific to branches of a decision tree.

All the best for your exams, and for the rest of the programme !