Python: High-Level (complexities are abstracted), interpreted, dynamically-typed (type information is stored with the data itself, allowing reintepretation at run-time), multi-paradigm, imperative, garbage collected. Extensive STL & third party libs, easy interface with other languages (C). Linking in Python is dynamic because types and bindings are determined at runtime.

Memory: Python Memory Manager (PMM). Stack for function calls + ref to objects. Heap for everything else. Python reserves memory pools, has greater overhead than C++, needs to store object-meta information, type, ref-count. Values are **Passed by object reference.**

Lists: Contiguous blocks of memory. Python over-allocates to cheapen appends/extends. Extend is cheaper than concat. Lists are multi-type. Removing is in-place, so expensive. Popping is cheap. itertools list of lists
Initialisation: `l = list()`, `l2 = []`
Append: `l.append(1)`, Concat: `l + l2`
Extend: `l.extend(l2)`, Contains: `x in l`
Remove: `l.remove(element), del l[index]`
Find (returns index of first match, or -1): `l.find(element)`
Sort (in-place, returns `None`): `l.sort(reverse=bool, key=lambda)`, Sum: `sum(l)`

Arrays: Elements of the same type, they're a class.
Import: `import array`
Initialisation: `int_array = array.array('i', [1, 2, 3, 4, 5])`

Stacks: FILO, used in DFS. Easiest way in python is to use lists.
Initialisation: `stack = []`
Push: `stack.append(element)`
Pop: `stack.pop()`
Peek: `stack[-1]`

Queues: FIFO, used in BFS. Can't use lists as expensive to remove first element. For a normal queue, use append and popleft.
Import: `from collections import deque`
Initialisation: `deque = deque()`
Append: `deque.append(element)`
AppendLeft: `deque.appendleft(element)`
Pop: `deque.pop()`, PopLeft: `deque.popleft()`
Remove (removes element): `deque.remove(element)`
Reverse: `deque.reverse()`
Rotate: `deque.rotate(right_rotate)`
Count (num instances of a value): `deque.count(value)`

Set: Unordered collection of elements. Hash-set. Can be initialised from any iterable type.
Initialisation: `set = set(), set2 = set([1, 2, 3])`
Append: `set.add(element)`
Remove: `set.remove(element)`
Intersection: `set.intersection(set2), set & set2`
Union: `set.union(set2), set | set2`
Complement: `set,difference(set2), set - set2`

HashMap: Can be initialised in list-comprehensions. Keys and values can vary in type, can be integer and string keys, uses elements `__hash__` function.
Initialisation: `map = dict()`
Contains: `x in map`, Set: `map[key] = value`, Get: `map[key]`
Keys: `map.keys()`, Values: `map.values()`, Both: `map.items()`

Strings: Immutable in Python
`list("hello")` gives a list of chars.
`'c'.isalnum()` is alpha-numeric.
`", ".join([strings])` concat list of strings with delimiter.

Binary:
Binary string (has "0b"): `b = bin(5444)`
To int (from base 2): `int(b, 2)`
`bit_i = a & 2 ** (i - 1) >> (i - 1)` or `a >> (i-1) & 1`

Heap (Priority-Queue): Tree-like data-structure, satisfies heap-invariant, for a min heap, each node is smaller than its children. Is a min-heap, define `__lt__` operator over elements.
Import: `import heapq`, Initialisation: `min_heap = []`
Push: `heapq.heappush(min_heap, element)`
Pop: `heapq.heappop(min_heap)`
Others: `heapq.heapify(x), heapreplace(heap, item), nlargest/smallest(n, heap)`

Enum:
```
from enum import Enum
class Color(Enum):
    Red=1
    Blue=2
```

Inheritance: Python handles function calls by traversing the class' method resolution order. `__mro__` class attribute. `__bases__` for immediate parents.

Abstract Base Classes: Classes inheriting ABC, have an ABCMeta class, which checks for `@abstractmethod` decorators and throws a `TypeError` if a class with an abstract method is instantiated.
```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area():
        pass
```

Object Equality:
```
def __eq__(self, other : object):
    if (isinstance(other, Base) and hasattr(other,
'value')):
        return self.value == other.value
    return False
```

Decorators: Wrap functions to provide additional functionality.
`@property, @celsius.setter`: Defines a method which behaves like an attribute, for additional get/set logic.
`@staticmethod`: Defines a static method.
`@classmethod`: Defines a class method.
`@functools.lru_cache(maxsize=1_000)`: Caches function outputs.

Iterators: Anything in Python which implements `__iter__` and `__next__` methods. (Duck typing). Can be iterated over.

Generators: Concise way of writing iterators. Returns on yield but stores state to continue execution when called again.

Concurrency: Async library, supports asynchronous (out-of-order) execution of a function. Similar to multi-processing in CPUs, interleaves execution of different calls. Co-routines can yield execution on a `await`.

Threading
```
import threading
thread = threading.Thread(target=method, args=(,))
thread.start()
```
`thread.join()` Join waits for thread to finish.

Pattern Matching:
```
point = Point(1, 2)
match point:
    case Point(x, y):
        print(f"Point at (x, y)")
    case _:
        print("Not point")
```

Exception Handling:
```
try:
except ZeroDivisionError:
else:
finally:
```

Exceptions:
`TypeError, ValueError, NotImplementedError, IndexError, KeyError, AttributeError, ZeroDivisionError, FileNotFoundError`

Interleaving String

```
from functools import cache
class Solution:
    def isInterleave(self, a:  str, b:  str, target:  str) -> bool:
        """
        Idea:  Dynamic programming on our index of a and b
        if current head of target matches either then consume and recurse
        """
        n, m = len(a), len(b)
        @cache
        def dp(i:  int, j:  int) -> bool:
            if i == n and j == m:
                return True
            return any([
                i < n and a[i] == target[i + j] and dp(i + 1, j),
                j < m and b[j] == target[i + j] and dp(i, j + 1),
            ])
        return n + m == len(target) and dp(0, 0)
```

Running Median

from typing import List

class RunningMedian():

@staticmethod def $_{binary_{s}earch(arr:List[int],x:int)->int:Returnstheindexofthefirstelementgeqtoxleft=0right=len(arr)while(left<right):midpoint=(le}$

def $_{init_{(self):self.store=[]}}$

def process$_{e}lem(self, elem) : index_{o}f_{i}nserted_{e}lem = RunningMedian._{binary_{s}earch(self.store,elem)self.store=self.store[:index_{o}f_{i}nserted_{e}lem}$

def get$_{m}edian(self) : n = len(self.store)if(n == 0) : raiseValueErrorif(nreturnself.store[n//2]return0.5*(self.store[n//2]+ self.store[(n//2) - 1])$

a = RunningMedian() for i in range(10): a.process$_{e}lem(i)print(a.get_{m}edian())$

Supervised Learning:

Classification:
`df['class'], classes = pd.factorize(df['class'])`
Input/Output Split:
```
input = df.drop("class", axis=1)
output = df[:, "class"]
```

Weighted Sampling
```
weights = 1 / df["class"].value_counts()
sample_weights = df["class"].map(weights)
```
With Pandas:
```
sample = df.sample(n, replace=bool, weights = nd.array)
```
With Jax
```
sample_indexes = jax.random.choice(key, a, shape=(), replace=True, p=weights)
samples=df.iloc[sample_indexes, :]
```

Initializers: Initialise the parameters of our model.
```
import flax.linen as nn
import nn.initializers as initializers
initializer = nn.initializers.lecun_normal()
params=initializer(rng, ((shape)), dtype=jnp.float64)
```
Others: `lecun_uniform`, `he_normal`, `he_uniform`

Optimizers: Numerical optimising schema.
```
import optax
optimizer = optax.adam(learning_rate)
optimizer_state=optimizer.init(params)
```
Updates:
```
loss, grads = jax.value_and_grads(loss_func, argnums=(,))(params, batch_in, batch_out)
updates, optimizer_state = optimizer.updates(grads, optimizer_state)
params = optax.apply_updates(params, updates)
```
Chaining:
```
op = optax.chain(optax.clip(1.0), optax.adam(10e-3))
```

Layers:
Linear: nn.Dense(features : int, kernel_init : nn.initializer)

DL Functions
```
from jax.nn import softmax, sigmoid, relu, onehot
onehot(data, num_classes=i)
```

Dropout
```
dropout_mask = jax.random.bernoulli(rng, shape=outputs.shape, p=1-dropout_rate)
outputs = outputs * dropout_mask / (1 - dropout_rate)
```

Pandas

Traversing OS:
```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Creating DataFrames
From List:
`pd.DataFrame(list, columns = ["student_id", "age"])`
From csv: `pd.read_csv(path, index_col=str)`

NANs
Fill (col_name: value): `df.fillna({"quantity": 0})`
Drop: `df.dropna(subset=["col_name"])`

DF Manipulations
Change column name: (dictionary, old-names to new-names):
`df.rename(columns : Dict[str, type], inplace=bool)`
Change column type (dictionary, column, type):
`df.astype({"col", int}, errors="raise"/"ignore")`
Concatenate: `pd.concat([df1, df2])`
Pivot (new index column): `df.pivot_table(index="month", columns=["city"], values="temperature", aggfunc="max")`

Sort Index: `df.sort_index(axis=1, ascending=bool)`
Sort Values: `df.sort_values(by=[col_names], ascending=bool)`

Data Split (stratify for classification)
```
from sklearn.model_selection import train_test_split
train_df, test_df = train_test_split(df, test_size = 0.1, stratify = df["class_index"])
```

Confusion Matrix `from sklearn.metrics import confusion_matrix, precision_score, recall_score, accuracy_score`
`fun(y_true, y_pred, average="micro"/"macro")`

Train Test Split samples = []

```
for class_index in range(num_classes):
    class_df = df[df["class_index"] == class_index]
    sample = class_df.sample(n = int(len(class_df) * proportion))
    samples.append(sample)
test_df = pd.concat(samples)
train_df = df[ df.index.isin(test_df.index)]
test_df.reset_index(drop=True, inplace=True)
train_df.reset_index(drop=True, inplace=True)
return train_df, test_df
```

Confusion Matrix

```
def confusion_matrix(params, batch_in, batch_out):
    confusion_matrix = np.zeros((10, 10))
    predictions = model.apply(params, batch_in)
    predicted_classes = jnp.argmax(predictions, axis=1)
    for (pred, actual) in zip(predicted_classes, batch_out):
    confusion_matrix[pred, actual] += 1
    # Micro-weights are the amount of each class (sum of column counts) / total
    micro_weights = jnp.sum(confusion_matrix, axis=0) / len(batch_out)
    # Precision is diag/row sum
    precisions = jnp.diag(confusion_matrix) / jnp.sum(confusion_matrix, axis = 1)
    # Recall is diag/col_sum
    recalls = jnp.diag(confusion_matrix) / jnp.sum(confusion_matrix, axis = 0)
    print(f"Marco-precision:  jnp.mean(precisions)")

    print(f"Marco-recall:  jnp.mean(recalls)")
    print(f"Marco-precision:  jnp.mean(precisions * micro_weights)")
    print(f"Marco-recall:  jnp.mean(recalls * micro_weights)")
    print(f"Accuracy:  jnp.sum(jnp.diag(confusion_matrix))/ jnp.sum(confusion_matrix)")
```

Data Parsing

```
import os
import pandas as pd
```

Simple Parse:

```
route = os.path.join(directory, file)
df = pd.read_csv(route, index_col : str)
```

Parse all in directory:

```
dfs = []
for file in os.listdir(directory):
    if (file.split('.')[-1] == "csv"):
        route = os.path.join(directory, file)
        dfs.append(pd.read_csv(route, index_col : str)
df = pd.concat(dfs)
```

Recursive Parse:

```
dfs = []
for root, dirs, files in os.walk(directory)
    if (file.split('.')[-1] == "csv"):
        route = os.path.join(root, file)
        dfs.append(pd.read_csv(route, index_col : str)
df = pd.concat(dfs)
```

Dataset Cleaning

Visualise NANs in each feature

```
print(df.isna().sum(axis=0))
```

Drop NANs, can specify which columns to apply to

```
df = df.dropna(subset : List[str])
```

Replace NANs

```
df.fillna(Dict[col_name, dtype]), df.fillna({"col_1": 2})
```

Data Formatting

Change column name: (dictionary, old-names to new-names):

```
df.rename(columns : Dict[str, type], inplace=bool)
```

Change column type (dictionary, column, type):

```
df.astype({"col", int}, errors="raise"/"ignore")
```

Output class indexing

```
df['class'], classes = pd.factorize(df['class'])
```

Tokenising Words

```
from transformers import BertTokenizer, FlaxBertModel
import jax.numpy as jnp

tokeniser = BertTokenizer.from_pretrained('bert-base-uncased')
embedding_model = FlaxBertModel.from_pretrained('bert-base-uncased')

df['sentence'] = df['sentence'].apply(lambda x : tokeniser.encode(x, add_special_tokens=True))
```

Split

```
from sklearn.model_selection import train_test_split
```

```
train_df, test_df = train_test_split(df, test_size = int(df.shape[0] * 0.3), stratify = df['class'])
```

Think about model inputs and structure

Batching & Processing

```
def sample(df : pd.DataFrame, batch_size : int):
    frequencies = 1.0 / df['class'].value_counts()
    weights = df['class'].map(frequencies)
    sample = df.sample(batch_size, replace : bool, weights = weights)
    return process_df(sample.copy())

def process_df(df : pd.DataFrame):
    batch_inputs = sample.drop(['class'])
    embeddings_output = embedding_model(batch_inputs)
    embeddings = embeddings_output.last_hidden_state
    batch_outputs = nn.one_hot(sample, num_classes : int)
    return jnp.array(embeddings), jnp.array(batch_outputs)
```

Model

```
import flax.linen as nn
import jax.nn.initializers as initializers

class Model(nn.Module):
    rate : float
    @nn.compact
    def __call__(self, x, inference : bool):
        x = nn.Dense(features : int, kernel_init = initializers.lecun_normal())(x)
        x = nn.Dropout(rate = self.rate)(x, deterministic=inference)
        x = nn.Conv(features=64, kernel_size=(3, 3), padding = "SAME", kernel_init = lecun_normal())(x)
        x = nn.relu(x)
        x = nn.max_pool(x, window_shape=(2, 2), strides=(2, 2))
        causal_mask = jnp.array([[j < i for j in range(x.shape[1])] for i in range(x.shape[0])])
        x = nn.MultiHeadDotProductAttention(num_heads=8, qkv_features=self.d_model, mask = casual_mask)(x)
        x = nn.LayerNorm()(x)

model.apply(params, x, rngs={'dropout': dropout_rng})
```

Cross Attender

```
class CrossAttention(nn.Module):
    d_kq : int
    d_v : int

    @nn.compact
    def __call__(self, in, target):
        k = nn.Dense(features = self.d_kq)(target)
        v = nn.Dense(features = self.d_v)(target)
        q = nn.Dense(features = self.d_v)(in)

        weights = softmax(q@k.T / sqrt(d_kq)) * mask
        return weights @ v
```

Positional Encoder

```
class PositionalEncoder(nn.Module):

    @nn.compact
    def __call__(self, x):
```

```
        num_data, num_features = x.shape
        frequencies = jnp.array([[pos/(10_000 ** (2 * feat / num_features)) for feat in range(num_features)] for
pos in range(num_data)])
        even_positions = jnp.sin(frequencies)
        odd_positions = jnp.cos(frequencies)
        positional_encodings = even_positions
        positional_encodings.at[:, 1::2].set(odd_positions[:, 1::2])
        return x + positional_encodings
```

Optimiser & Params

```
rng, init_rng = jax.random.split(jax.random.PRNGKey(42), 2)
example_batch_in, _ = sample(train_df)
params = model.init(init_rng, example_batch)

import optax
optimizer = optax.adam(learning_rate)
optimizer_state=optimizer.init(params)
```

Chaining:
```
op = optax.chain(optax.clip(1.0), optax.adam(10e-3))
```

Loss Functions

```
def mse(params, batch_in, batch_out):
    model_outputs = model.apply(params, x)
    return jnp.mean(jnp.square(model_outputs - batch_out)

def cross_entropy(params, batch_in, batch_outputs):
    model_outputs = model.apply(params, batch_inputs)
    jnp.mean(jnp.sum(batch_outputs * jnp.log(model_outputs), axis=1)
```

Training Loop

```
for epoch in range(num_epochs):
    for batch in range(train_df.shape[0] // batch_size):
        batch_inputs, batch_outputs = sample(train_df)
        loss, grads = jax.value_and_grads(loss_func, argnums=(0,))(params, batch_in, batch_out)
        updates, optimizer_state = optimizer.updates(grads, optimizer_state)
        params = optax.apply_updates(params, updates)
```

Confusion Matrix

Inputs are argmax:
```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, accuracy_score
fun(y_true, y_pred, average="micro"/"macro")
```