CONDITION MONITORING OF MARINE MACHINERY USING ARTIFICIAL INTELLIGENCE

Section A Introduction to Artificial Intelligence and applications

1. History of AI
2. Objectives of AI
3. Subfields of AI
4. Uses and application of AI

Section A Introduction to Artificial Intelligence and applications

1955- Allen Newell and Herbert A. Simon created "Logic Theorist"

1956 - John McCarthy coined the term ‘artificial intelligence’ and had the first AI conference

1974 to 1993 - Two AI Winters and a Boom in AI

1993 to 2011- The emergence of big data

2011 to present - Highly intelligent agents

Objectives of AI

1. Solve complex problems faster
2. Use lesser computation than conventional techniques for problem solving
3. Complete complex tasks and reduce the amount of time needed to perform specific tasks
4. Facilitate human-computer interaction

Subfields of AI

1. Machine learning
2. Deep learning
3. Natural language processing
4. Cognitive computing
5. Computer vision
6. Fuzzy logic

Use of AI

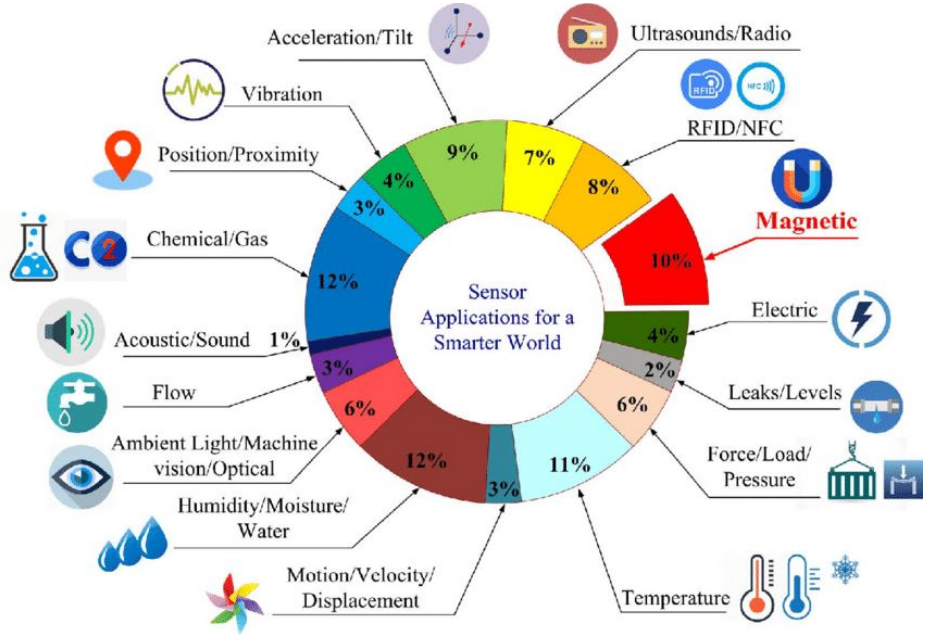
1. Used to reduce or avoid repetitive tasks.
2. To improve an existing product
3. Used in industries, from marketing to supply chain, finance, food-processing sector.

Section B- Condition Based Monitoring and scheduled monitoring

1. Sensors
2. Sensors used in marine machinery
3. Data collection
4. Condition monitoring
5. Sensors to check the health of a machine
6. Digital twin

Sensors

Produces an output signal for the purpose of sensing a physical phenomenon.



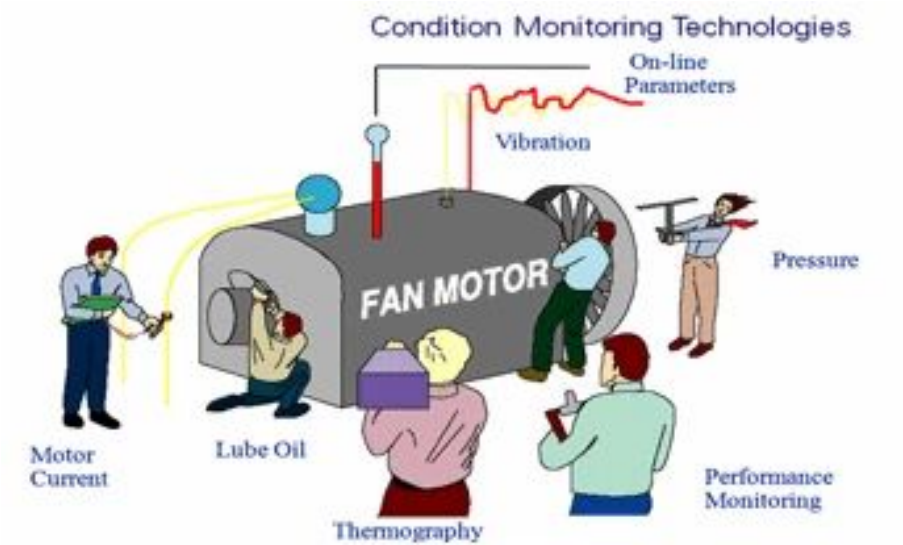
Source: Overview of Spintronic Sensors With Internet of Things for Smart Living. IEEE Transactions on Magnetics. PP. 10.1109/TMAG.2019.2927457.

Types of parameters for sensor measurement on maritime vessels

1. Vibration and acoustic
2. Temperature and thermography
3. Pressure
4. Tribology
5. Torque
6. Flow rates
7. Contamination
8. Power and speed typically associated with performance
9. Electricity (Voltage, Current, Frequency, Harmonics)

Checking the health of a machine

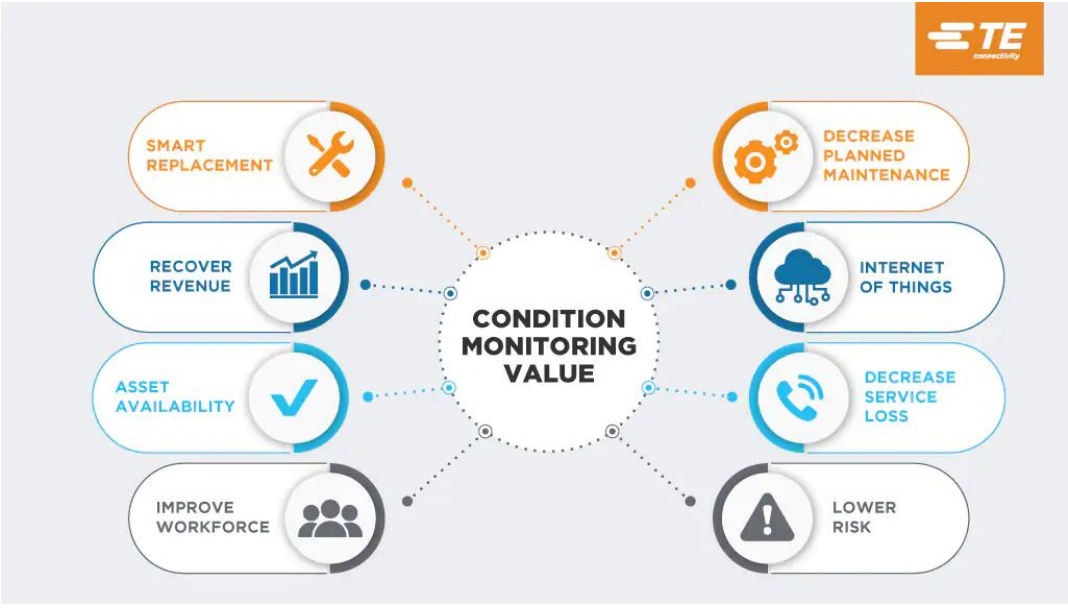
Sensors monitor for abnormalities in the system and based on past data they can correlate and estimate the current health of the system



Source: National Instruments (https://www.ni.com/en-in/innovations/white-papers/14/sensors-for-condition-monitoring.html)

Condition monitoring

process of monitoring the parameters contributing to the ‘health’ of a machinery in order to identify any abnormal changes from the standard values which can indicate a potential fault that is about to occur in the future.



Source:
https://www.te.com/usa-en/industries/sensor-solutions/applications/industrial-condition-monitoring-sensors.html

Condition monitoring

Using Condition monitoring, each sensor offers the ability to monitor the degradation of mechanical and electrical components within an equipment.

Some examples are as follows:

1. Vibration sensors are used to detect roller bearing wear, gearbox wear, shaft misalignment, unbalance, and mechanical looseness.
2. Speed sensors work with vibration sensors to correlate vibrations to rotating speed and shaft angular position.
3. Motor current sensors are commonly placed at the motor control center. They can detect eccentric rotors, loose windings, rotor bar degradation, and electrical supply unbalance.
4. Dynamic pressure sensors are used for combustion dynamics, flow turbulence, and cavitations.
5. Temperature sensors are typically used to detect heat caused by friction. They often accompany vibration sensors to collaborate vibration-detected degradation.
6. Thermal imaging detects hundreds of temperatures within the camera's field of view.
7. Ultrasonic sensors can detect electrical problems including corona, arcing, and tracking. They can also be used to detect early signs of roller bearing wear.
8. Oil sensors can detect wear debris from bearings and gears. They also can detect contaminants in the oil that reduce the lubrication ability of the oil.

Digital Twin

- Virtual model designed to accurately reflect a physical object
  a. Gather real world data
  b. feed into the digital mode
  c. Simulate the performance
  d. Optimize the system
  e. Apply the new learning

Source: https://new.abb.com/news/detail/80770/the-digital-twin-from-hype-to-reality

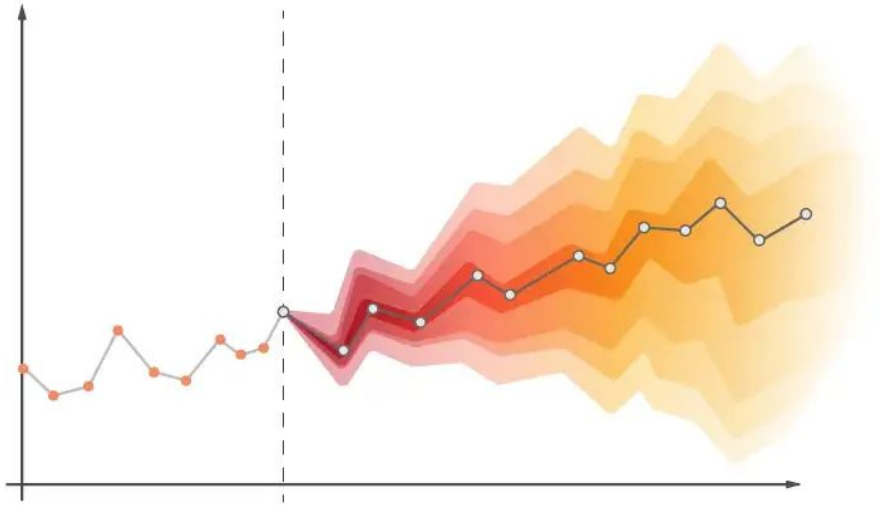
Section C-Forecasting

1. Forecasting
2. Classes of forecasting
3. Simple Moving average (SMA), Auto regression
4. How regression and MA can be used for forecasting
5. Data forecasting using Auto Regression
6. Data forecasting using ARIMA Models

Forecasting

Process of making predictions based on past and present data

Eg: Weather forecasts, Economic forecasts, Sales Forecasts etc.



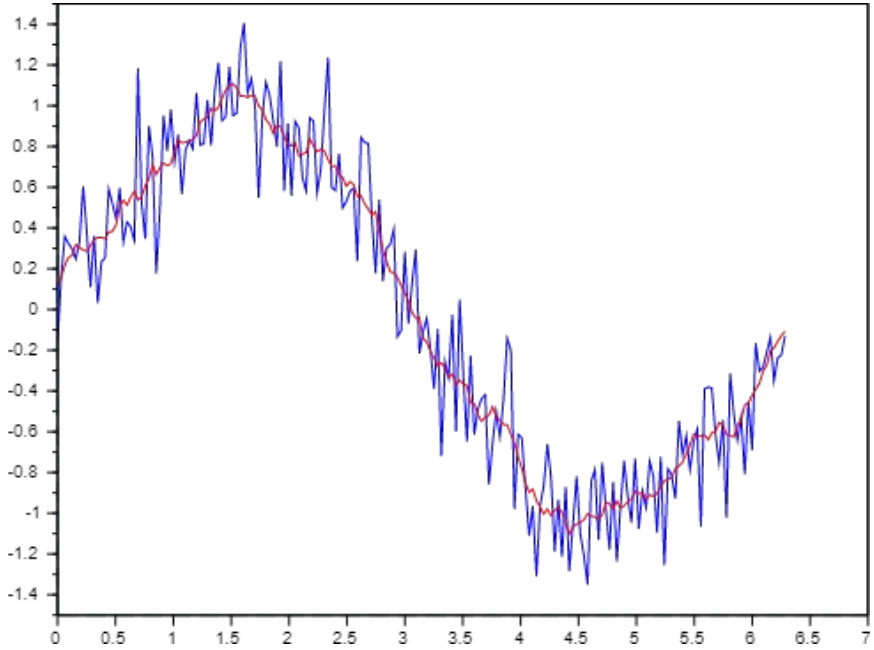
Classes of forecasting

1. Qualitative vs. quantitative methods
2. Average approach
3. Drift method
4. Time series methods
5. Relational methods
6. Judgmental methods
7. Artificial intelligence methods
8. Geometric Extrapolation with error prediction
9. Regression method

Simple Moving Average (SMA)

Takes the average of the last k points to predict the n^{th} point

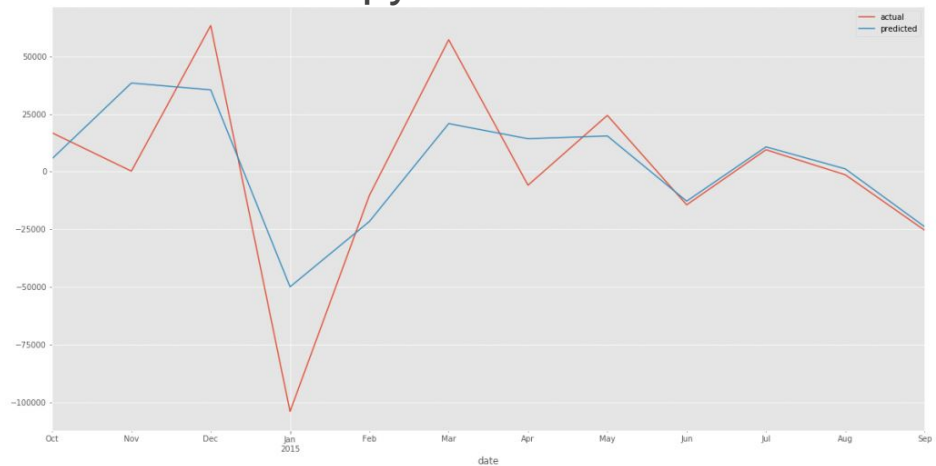
$$SMA_k = \frac{1}{k} \sum_{i=n-k+1}^{n} p_i$$



Source: https://en.wikipedia.org/wiki/Moving_average

Auto Regression

- operate under the premise that past values have an effect on current values, which makes the statistical technique popular for analyzing nature, economics, and other processes that vary over time.
- Other models use only a linear combination of predictors
- Open source codes available in python



Source: https://pythondata.com/forecasting-time-series-autoregression/

Autoregressive integrated moving average (ARIMA)

- Statistical analysis model that uses time series data to either better understand the data set or to predict future trends
- Uses delayed or lagged moving averages to smooth time series data
- Used to make time series data stationary
- Open source codes available in python

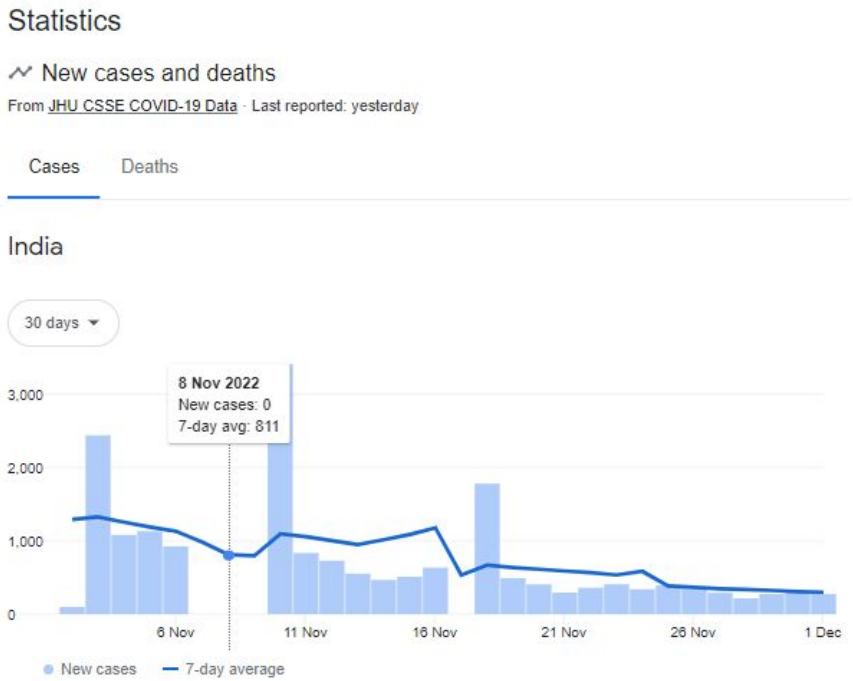
Section D-Bad data Detection

1. Bad data
2. Bad data detection
3. Methods used to detect bad data
4. The different types of graphs
5. Detecting bad data from graphs

Bad data

inaccurate set of information

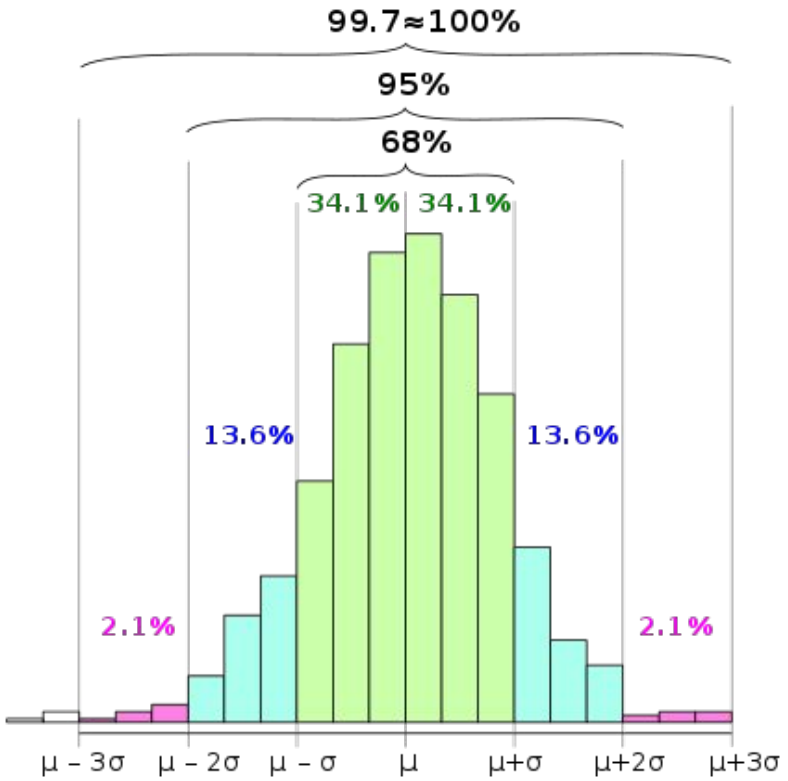
1. missing data
2. wrong information
3. inappropriate data
4. non-conforming data
5. duplicate data
6. poor entries (misspells, typos, variations in spellings, format etc).



Source: Google

Bad data detection

1. Median
2. Standard Deviation
3. 3 sigma rule
4. p-value test



Source: Wikipedia

Bad data detection

1. Median
2. Standard Deviation
3. 3 sigma rule
4. p-value test

Source: Wikipedia

Section E-Introduction to Python using Spyder-Anaconda package and practical case study

1. Use of Spyder(Anaconda) package(Open Source)
2. Syntax pertaining to Loops, Conditional statements, functions and file handling,
3. Opening and reading the contents of a file, opening and writing/appending data to a file
4. Plotting graphs
5. Case Study
6. Develop a fully fledged condition monitoring system for monitoring the health of a marine machine

Loops

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An `if` … `elif` … `elif` … sequence is a substitute for the `switch` or `case` statements found in other languages.

https://docs.python.org/3/contents.html

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword ‘`elif`’ is short for ‘else if’, and is useful to avoid excessive indentation. An `if` … `elif` … `elif` … sequence is a substitute for the `switch` or `case` statements found in other languages.

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see Looping Techniques.

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

4.4. `break` and `continue` Statements, and `else` Clauses on Loops

The `break` statement, like in C, breaks out of the innermost enclosing `for` or `while` loop.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the iterable (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, **not** the `if` statement.)

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see Handling Exceptions.

The `continue` statement, also borrowed from C, continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`clear`**()
Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**(*x*[, *start*[, *end*]])

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

`list.`**`count`**(*x*)

Return the number of times *x* appears in the list.

`list.`**`sort`**(*, *key=None*, *reverse=False*)

Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

`list.`**`reverse`**()

Reverse the elements of the list in place.

`list.`**`copy`**()

Return a shallow copy of the list. Equivalent to `a[:]`.

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and *None* can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of `namedtuples`). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

7.2. Reading and Writing Files

`open()` returns a file object, and is most commonly used with two positional arguments and one keyword argument: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific *encoding*. If *encoding* is not specified, the default is platform dependent (see `open()`). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know that you need to use a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as `bytes` objects. You can not specify *encoding* when opening file in binary mode.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Warning: Calling `f.write()` without using the `with` keyword or calling `f.close()` **might** result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* characters (in text mode) or *size* bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *whence* argument. A *whence* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *whence* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid *offset* values are those returned from the `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

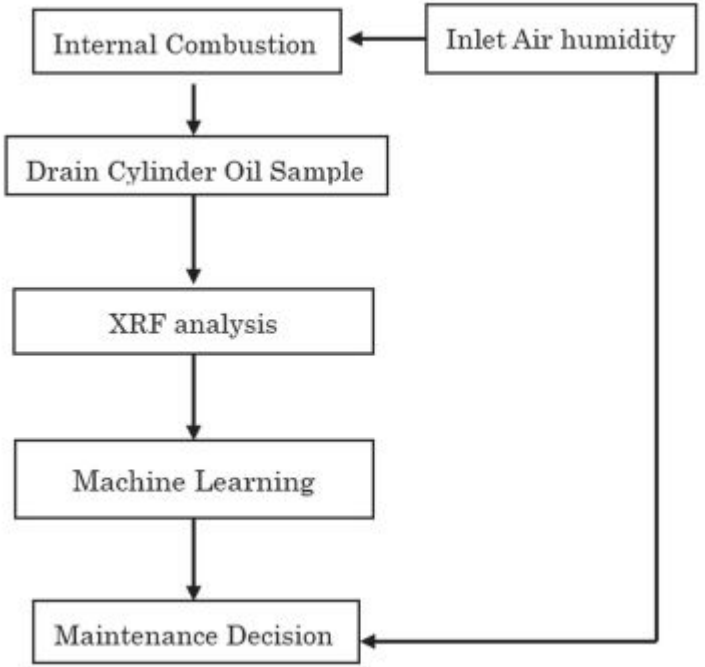
File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

Case Study 1- Condition-Based Monitoring for Marine Engine Maintenance by Analyzing Drain Cylinder Oil Sample

6UEC60LSA engine Specifications

Engine model	6UEC60LSA
Bore (mm)	600
Stroke (mm)	1,600
Output (kW)	11,180
Engine speed (rpm)	110
Piston speed (m/s)	7.09
Engine length (mm)	7,270
Piston overhaul height (mm)	7,700
Crankshaft center (mm)	930
Bedplate width (mm)	3,000
Engine weight (Ton)	239

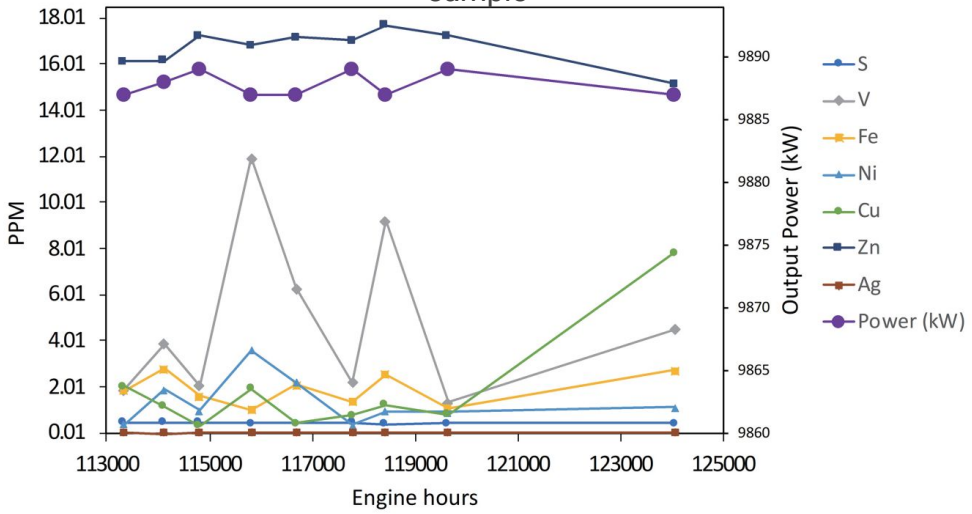
Maintenance decision making framework



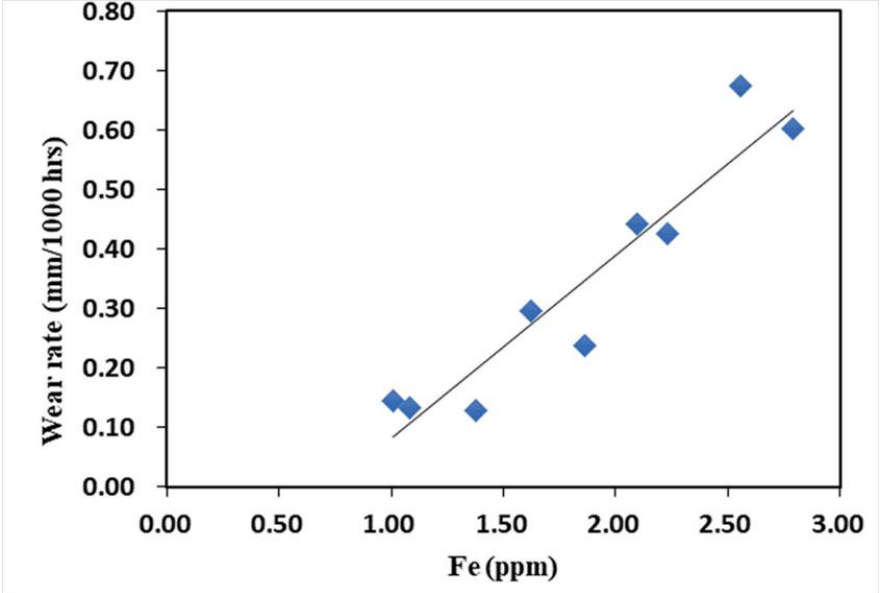
Famakinwa, Ayo, and Tadahiro Shibutani. "Condition-Based Monitoring for Marine Engine Maintenance by Analyzing Drain Cylinder Oil Sample." Tribology Online 17.2 (2022): 71-77.

Case Study 1

Engine output and particles quantity in oil sample



Graphical representation of average wear rate



Famakinwa, Ayo, and Tadahiro Shibutani. "Condition-Based Monitoring for Marine Engine Maintenance by Analyzing Drain Cylinder Oil Sample." Tribology Online 17.2 (2022): 71-77

Case Study 2- Sensors Specifications For Maritime Condition Monitoring Based On Failure Mode, Effects and Criticality Analysis (FMECA)

Sensor selection is an important step during the Condition Monitoring process. The factors affecting the selection of a sensor can be as follows:

1. Determine the variables to be measured
2. Determine the technical specification of sensors for each measurement
3. Determine the availability and affordability of sensors
4. Determine the installation, maintenance plan and calibration procedure of sensors

Shan Guan, Knut Erik Knutsen, and Øystein Åsheim Alnes, " Condition Monitoring Based On Fmeca: A Case Study Of Sensors Specifications For Ma
proceedings of MFPT 2017, Society for Machinery Failure Prevention Technology

Case Study 2

Component	Failure Modes	Failure Effects	Failure Criticality	Sensors
Electric Motor	Overheating	Short motor life to motor failure	High to Very High	Temperature
Frequency Converter	Temperature induced	Components/ system failure, Won't start	High to Very High	Thermography
Shaft	Shaft failure	Components/ System failure	High to Very High	Vibration, Acoustic Emission, UT
	Sheared Shaft, Shaft failure	Seized	High to Very High	Vibration, Acoustic Emission, UT
Tooth Coupling	Teeth wear away	High vibration to system failure	Medium to Very High	Vibration, Torque, Particulate analysis/ Wear Debris
	Tooth fatigue failure	High vibration to system failure	High	Vibration, Torque

Case Study 2

Rolling Bearing	Rolling contact fatigue	Seized to system failure	High to Very High	Vibration, Temperature, Oil analysis (off site)/ Wear Debris
	Plastic deformation	Noisy/Excessive vibration, Seized Motor, Loss of torque	Medium to Very High	Vibration
Bevel Gear	Plastic deformation	Noisy, Vibration, System Failure	Medium to Very High	Oil analysis (off site)/ Wear Debris, Vibration
	Tooth flank contact fatigue	Vibration, System Failure	Medium to Very High	Oil analysis (off site)/ Wear Debris, Vibration
Propeller Blade	Fatigue failure	Loss of Torque, Vibration, System Failure	Medium to Very High	Torque, Vibration, Ultrasonic
Lubrication System	Pressure drop	Components/System failure in long term	Medium to Very High	Oil Pressure
	Overheating	Components/System failure in long term	Medium to Very High	Temperature

Case Study 2

	Accelerometers	**Velocity Vibration Sensors**	**Displacement Vibration Sensors**
Measuring Parameters	Acceleration	Velocity	Displacement
Sensing Mechanism	Piezoelectric Sensors	Electromagnetic transducer	Capacitance sensors or Eddy-current probe
Major Advantages	Good response at high frequencies; Able to stand high Temperature; Small size	Good response in middle range frequencies; Able to stand high temperature; Low Cost; No external power needed.	Non-contact, No wear; Able to measure both static and dynamic displacements; Good response at low frequencies
Major Disadvantages	Sensitive to high frequency noise; Higher cost	Low resonant frequency and phase shift; Large footprint; Cross noise	Bounded by high frequencies; Sensitive to Electrical and mechanical noise
Wireless Capability	Commercially available	Not available	Possible, not yet commercially available.
MEMS-based devices available	Commercially available	Not available	Commercially available

Case Study 2

Tech Specification	SLD144S Vibration Sensor	General Vibration Sensors for Ship Machinery CM
Nominal sensitivity, main axis	100 mV/g	10-500 mV/g
Transverse sensitivity	Max. 10%	Max. 10%
Typical base strain sensitivity	0.01 m/s2/μ strain	0.01-0.05 m/s2/μ strain
Linear frequency range	2 Hz - 10 kHz (±1dB) (–3 dB at 0.7 Hz)	2 Hz - 10 kHz (±1dB) (–3 dB at 0.7 Hz)
Max. peak acceleration	600 m/s2 = 60 g	100 g
Settling time	3 sec	1-5 sec
Bias point	11 to 13 V (typical 12 V)	11 to 13 V (typical 12 V)
Temperature range	40° C to +125° C	0° C to +175° C
Power requirements	24 V /2 - 5 mA	5-24 V (1-10 mA)
Casing	Stainless acid proof steel	Marine Environment Resist materials preferred
Isolation	Case isolated, > 1 Mohm	Case isolated, > 1 Mohm