Python + debuging and profiling

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

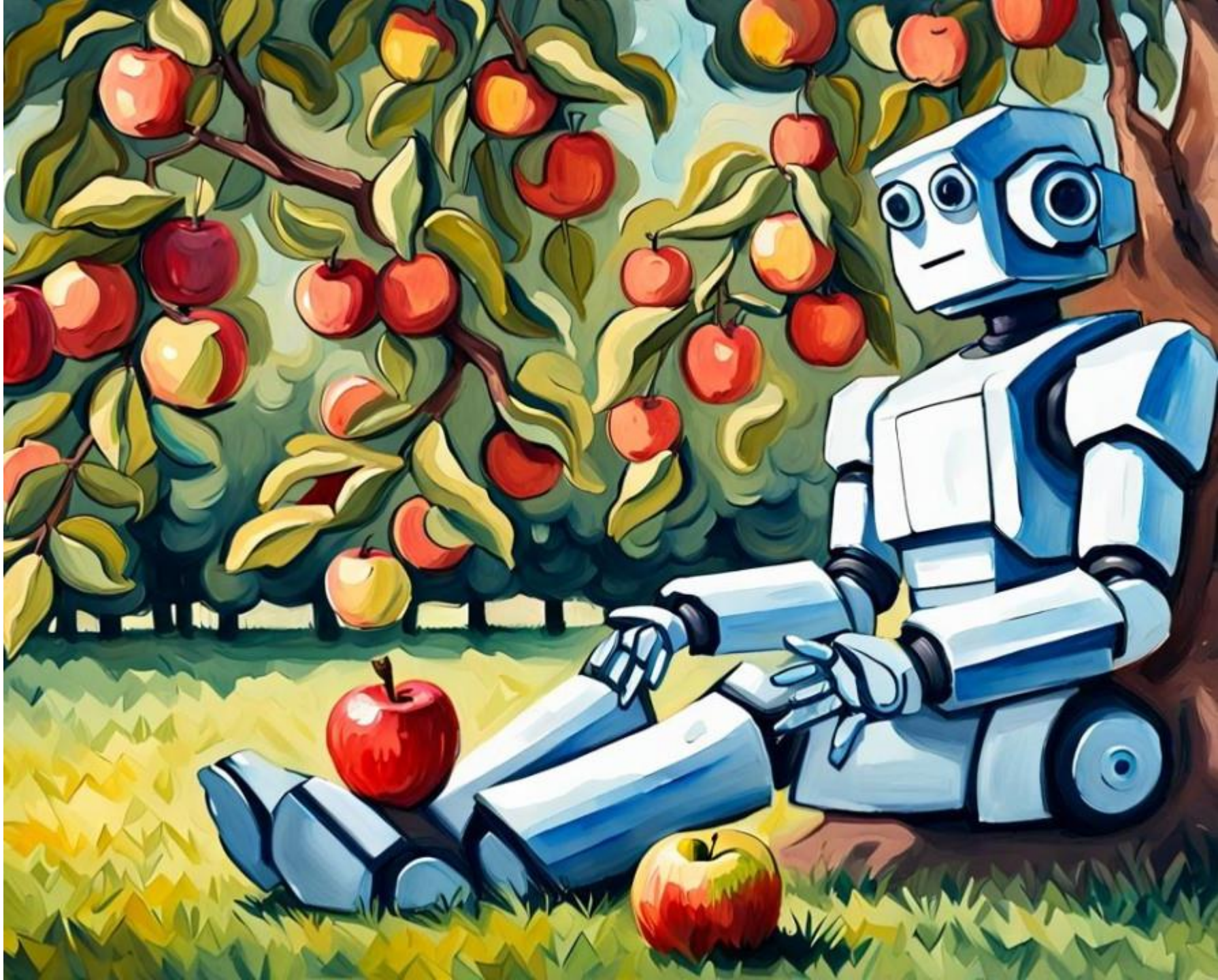
Python

debugging

Profiling

?

Python



What is Python?

Python is a **high-level**, **general-purpose**, **interpreted** programming language known for its simplicity and readability. It is widely used for web development, data science, automation, machine learning, artificial intelligence, and more. Python is an interpreted language, meaning the code is executed line-by-line by an interpreter. It's versatile, easy to learn, and supports a wide range of applications across various domains.

- **High-level Language**: Python is easy to learn and doesn't require deep knowledge of computer internals.
- **General-purpose**: Python is used in various fields like web development, data science, machine learning, automation, testing, and more.
- **Interpreted**: Python code is executed line-by-line by an interpreter, converting it to machine code as it runs.

Hello Word

- Python scripts
  Hello.py:

```
print(”Hello, Word!”)
```

  Run:

```
$python Hello.py
Hello, Word!
```

- Python REPL (Read-Eval-Print Loop)

```
$python
>>> print(”Hello, Word!”)
Hello, Word!
```

- Jupyter
  Google Colab
- IDEs
  VS Code
  PyCharm

Variables

- **Variables** in Python are dynamically typed, so you don't need to declare their type explicitly.

```
x = 5
y = 3.5
x = True
```

- Python supports many **data types** such as `integers`, `floats`, `strings`, `lists`, `dictionaries`, and more.

Types

- Integer
  x = 10
- Float
  ```
  x = 3.14
  ```
- String
  ```
  name1 =”Bob”
  name2 = ’Alice’
  names = name1+name2
  #BobAlice
  ```
- Bool
  ```
  odd = True
  odd = False
  ```

Types

- List - An ordered, mutable collection

```
np = ['zero', 1, 2, 3]
np[0] = 0
np[-1] = 4
print (np)
[0, 1, 2, 4]
```

- Tuble - An ordered, immutable collection

```
np = ('zero', 1, 2, 3)
```

- Dictionary - key-value pairs

```
person = {"ID":123, "name":"Bob"}
```

- Set - A collection of unique, unordered items

```
IDs = {123, 234, 554}
```

Variables

- You can **check and convert types** using functions like `type()` and `int(), str(), etc.`

```
>>>x = ”42”
>>>print(type(x))
<class 'str’>
>>>y = int(x)
>>>print(type(y))
<class 'int'>
```

- **Mutable** types (e.g., `lists, dictionaries`) **can** be changed,
- **Immutable** types (e.g., `strings, tuples`) **cannot** be altered after creation.

Control flow

```
x = 10
if x > 5:
    print("x is greater than 5")
    big = True
else:
    print("x is 5 or less")
```

Control flow

```
x = 10
if x > 5:
    print("x is greater than 5")
    big = True
else:
    print("x is 5 or less")
```

Blocks Work in Python:

- **Indentation** is used to define the beginning and end of blocks of code.
- Blocks of code are typically used in **functions**, **loops**, **conditionals**, **classes**, etc.
- Each level of indentation indicates a new block, and returning to the previous indentation level ends the current block.
- The block is executed as a unit when the control flow reaches it.

loops

```
nums = range(5)
for i in nums:
    print(i)
```

```
1
2
3
4
5
```

```
Names = (”Bob”, ”Alice”)
for name in Names:
    print(name)
```

```
Bob
Alice
```

Function

```
def greet(name):
    return "Hello" +name + "!"

print(greet("Alice"))
#Hello, Alice!
```

Packages

- **Modules** are simply Python files (.py) that contain functions, classes, and variables.
- You can **import** modules into other Python scripts to reuse code.

```
import numpy
import numpy as np
from numpy import pi
```

- **Packages** are directories that contain multiple modules.
- **Built-in modules** provide standard functionality and can be accessed by importing them.

packages

- **Data Science and Machine Learning**
  - NumPy
  - Pandas
  - SciPy
  - Scikit-learn
  - TensorFlow
  - PyTorch
  - Statsmodels
- **Web Development**
  - Flask
  - Django
  - FastAPI
- **Automation and Scripting**
  - Selenium
  - PyAutoGUI
  - Paramiko
  - Fabric
- **Data Visualization**
  - Plotly
  - Matplotlib
  - Seaborn
- **Testing**
  - Unittest
  - pytest

Files

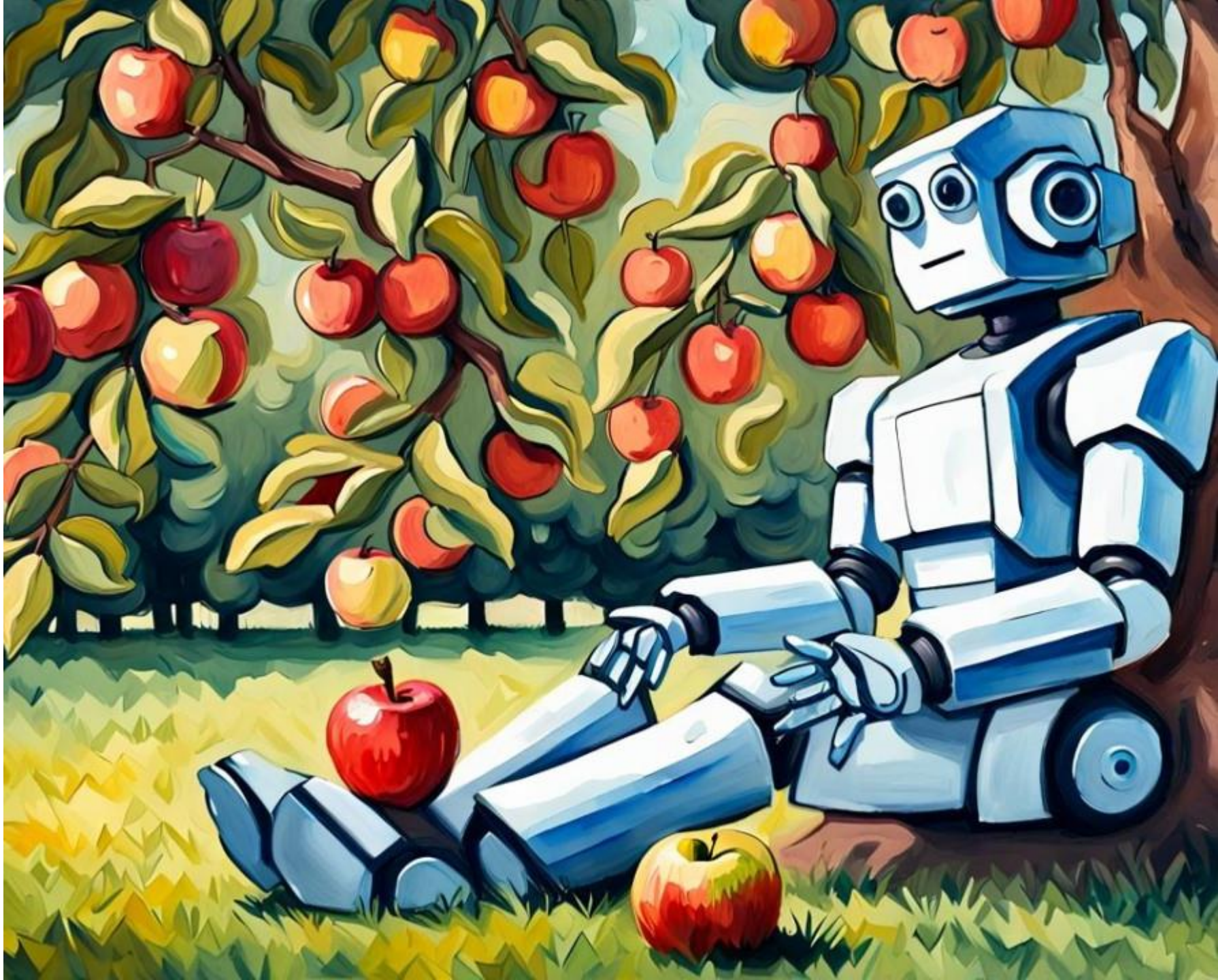
- Write to a file

```
with open("example.txt", "w") as file:
    file.write("Hello, File!")
```

- Read from a file

```
with open("example.txt", "r") as file:
    print(file.read())
#Hello, File!
```

Debugging



Print

The most basic debugging technique is to insert `print()` statements in your code to check the flow and variable values.

```
def add(a, b):
# Debugging line
    print(a, b)
    return a + b

result = add(3, 4)
print("Result: " + result)
```

pdb

- The pdb module is Python's built-in debugger. It allows you to set breakpoints, step through code, and inspect variables.
- To pause the execution of your program at a certain line, use pdb.set_trace():
  import pdb

  def example_function(a, b):
    result = a + b
  pdb.set_trace() # Pause here for debugging
  return result

  example_function(5, 10)
- run the script using the -m pdb flag:
  python -m pdb my_script.py

pdb

You can use several commands to inspect and control the program.

- **`n (next)`:** Execute the next line of code (step over functions).
- **`s (step)`:** Step into functions, if applicable.
- **`c (continue)`:** Continue execution until the next breakpoint is hit.
- **`q (quit)`:** Exit the debugger and stop the program.
- **`p (print)`:** Print the value of a variable or expression
- **`l (list)`:** Show the current line of code and a few lines before/after it.
- **`b (breakpoint)`:** Set a breakpoint at a specific line (line number or function)

logging

The logging module allows you to log messages at different levels (debug, info, warning, error, etc.) and is a more robust alternative to `print()` for debugging in larger projects.

```
import logging
# Set up logging configuration
logging.basicConfig(level=logging.DEBUG)

def add(a, b):
    logging.debug(f"a: {a}, b: {b}")  # Log the values
of variables
    return a + b

result = add(3, 4)
logging.info(f"Result: {result}")
```

logging

Levels of Logging:

- `DEBUG`: Detailed information, typically useful for diagnosing problems.
- `INFO`: General information about program execution.
- `WARNING`: Indicates something unexpected but doesn't stop the program.
- `ERROR`: Something went wrong, and the function didn’t complete properly.
- `CRITICAL`: A very serious error that prevents the program from running.

try except

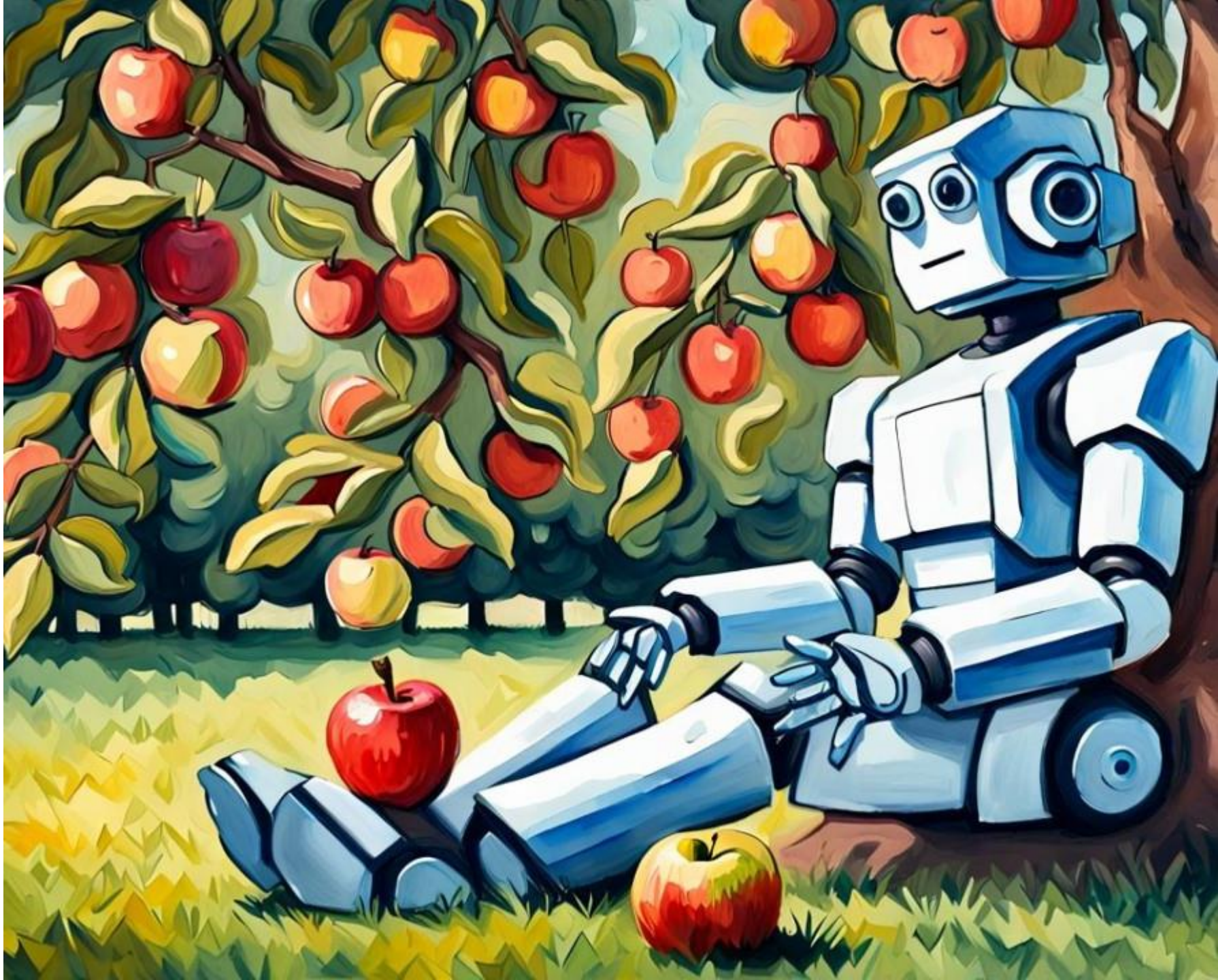
Instead of crashing your program, you can handle exceptions to log or print meaningful error messages.

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("Error: " + e)
```

Debugging

	Pros	Cons
print	Simple and easy to use.	Can clutter code with unnecessary prints.
	Quick to implement for small issues.	Difficult to track the flow in larger programs.
pdb	Allows interactive debugging with breakpoints.	Requires inserting breakpoints manually.
	Can step through code and inspect variables.	Not as intuitive for beginners compared to IDE debuggers.
	Powerful for complex debugging.	Doesn’t allow real-time code execution control.
Try-Except	Prevents program crashes from runtime errors.	Doesn’t help with logic errors or issues that don’t throw exceptions.
	Can provide meaningful error messages to the user.	Overuse can hide underlying issues if not used properly.

Profiling



Profiling

- Code profiling is the process of analyzing a program's execution to measure performance-related aspects like execution time, memory usage, and function call frequency. Profiling helps developers **identify bottlenecks**, optimize code, and improve efficiency.

Types of Code Profiling

- **CPU Profiling** – Measures how much time each function takes.
- **Memory Profiling** – Tracks memory allocation to detect leaks.
- **Line-by-Line Profiling** – Analyzes execution at the line level.
- **Real-Time Profiling** – Monitors performance while the program runs.

Why is profiling important?

- Identifies Performance Bottlenecks - Finds the slowest functions and inefficient code.
- Optimizes Code for Speed – Helps choose faster algorithms and data structures.
- Reduces Memory Usage - Detects memory leaks and unnecessary object allocations.
- Prevents Unnecessary Computations - Shows how many times each function is called.
- Improves Scalability - Ensures code can handle large datasets and high traffic.
- Debugs Unexpected Issues - Detects infinite loops, recursion errors, and inefficient memory usage.
- Saves Time and Costs
  - Reduces cloud/server expenses by optimizing CPU & RAM usage.
  - Speeds up development by reducing debugging time.
  - Improves user experience with faster response times.

bash + time

The `time` command in Linux measures the real (wall clock) time, user CPU time, and system CPU time taken to execute a program.

```
time python script.py
```

Output:

```
real 0m1.234s     <- Total elapsed time (wall-clock time).
user 0m1.200s     <- CPU time spent in user mode.
sys 0m0.034s      <- CPU time spent in kernel mode.
```

time

Using time Module To measure the elapsed time for specific parts of the code::

```
import time
start_time = time.time()
# Your Python code here
time.sleep(2)
# Simulate some processing
end_time = time.time()
print("Elapsed time: " +
    (end_time - start_time) + " seconds")
```

timeit

- $python -m timeit "'-'.join(str(n) for n in range(100))„
  10000 loops, best of 5: 30.2 usec per loop
- $python -m timeit "'-'.join([str(n) for n in range(100)])„
  10000 loops, best of 5: 27.5 usec per loop
- $python -m timeit "'-'.join(map(str, range(100)))„
  10000 loops, best of 5: 23.2 usec per loop

- import timeit
  timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
  0.3018611848820001
  timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
  0.2727368790656328
  timeit.timeit('"-".join(map(str, range(100)))', number=10000)
  0.2370267980732023 7

cProfile

The *cProfile* module is a built-in Python library for profiling. It provides detailed information about function calls and execution time.

- It gives you the **total run time** taken by the entire code.
- It also shows the **time taken by each individual step**. This allows you to compare and find which parts need optimization
- cProfile module also tells the **number of times certain functions are being called**.
- The data inferred can be **exported easily** using pstats module.
- The data can be visualized nicely using snakeviz module.

cProfile

- Profiling a Script
  Run your script with cProfile from the command line:
  ```
  python -m cProfile -s time your_script.py
  ```
- Profiling Specific Code
  You can also use cProfile in your script:
  ```
  import cProfile

  def my_function():
      # Your code here
      pass

  cProfile.run('my_function()', sort='time')
  ```

Sorting Option	Description
calls	Sort by the number of calls
time	Sort by total time spent in each function
cumtime	Sort by cumulative time (including sub-functions)
name	Sort by function name
filename	Sort by filename
line	Sort by line number

```
6 function calls in 1.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.002    1.002 script.py:10(main)
        1    0.000    0.000    1.001    1.001 script.py:4(slow_function)
        1    0.001    0.001    0.001    0.001 script.py:8(fast_function)
        1    0.000    0.000    1.002    1.002 <string>:1(<module>)
        1    0.000    0.000    1.002    1.002 {built-in method builtins.exec}
        1    1.000    1.000    1.000    1.000 {built-in method time.sleep}
```

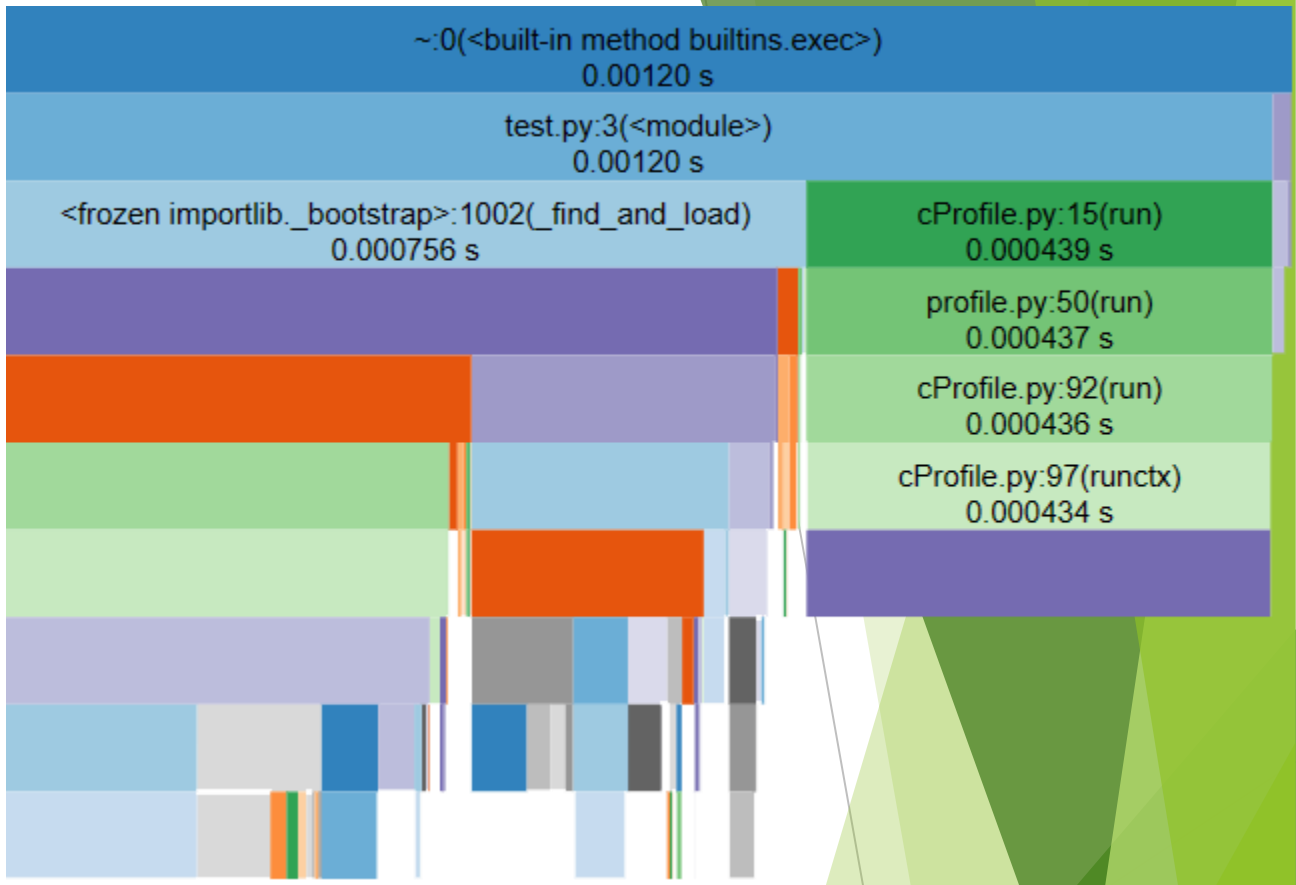
WithOUT sub-fun

WITH sub-fun

cProfile

```
import cProfile
import time

def slow_function():
    time.sleep(1)
    sum([i**2 for i in range(10000)])

def fast_function():
    sum([i**2 for i in range(100)])

def main():
    slow_function()
    fast_function()

if __name__ == "__main__":
    cProfile.run('main()')
```

cProfile + snakeviz



https://www.w3resource.com/python-interview/how-can-you-sort-and-visualize-cprofile-output-to-identify-performance-issues-easily.php

Memory-profile

- memory_profiler is a Python module used to measure memory usage line-by-line in a script. It is useful for identifying memory leaks and optimizing memory usage.
- `pip install memory-profiler`

Memory-profile

script.py :

```
from memory_profiler import profile
import numpy as np.

@profile
def memory_intensive_function():
    # Allocate a large list
    large_list = [i**2 for i in range(1000000)]

    # Allocate a large NumPy array
    large_array = np.ones((1000, 1000))

    return sum(large_list) + np.sum(large_array)

if __name__ == "__main__":
    memory_intensive_function()
```

Run profiler:

```
$python -m memory_profiler script.py
```

current

change

```
Filen      ot.py

Line# Mem usage    Increment Occurrences   Line Contents
=============================================================
4     12.2 MiB    12.2 MiB           1       @profile
5                                             def memory_intensive_function():
6     49.8 MiB    37.6 MiB           1           large_list = [i**2 for i in range(1000000)]
9     57.4 MiB     7.6 MiB           1           large_array = np.ones((1000, 1000))
11     57.4 MiB     0.0 MiB           1           return sum(large_list) + np.sum(large_array)
```

Memory-profile

script.py:

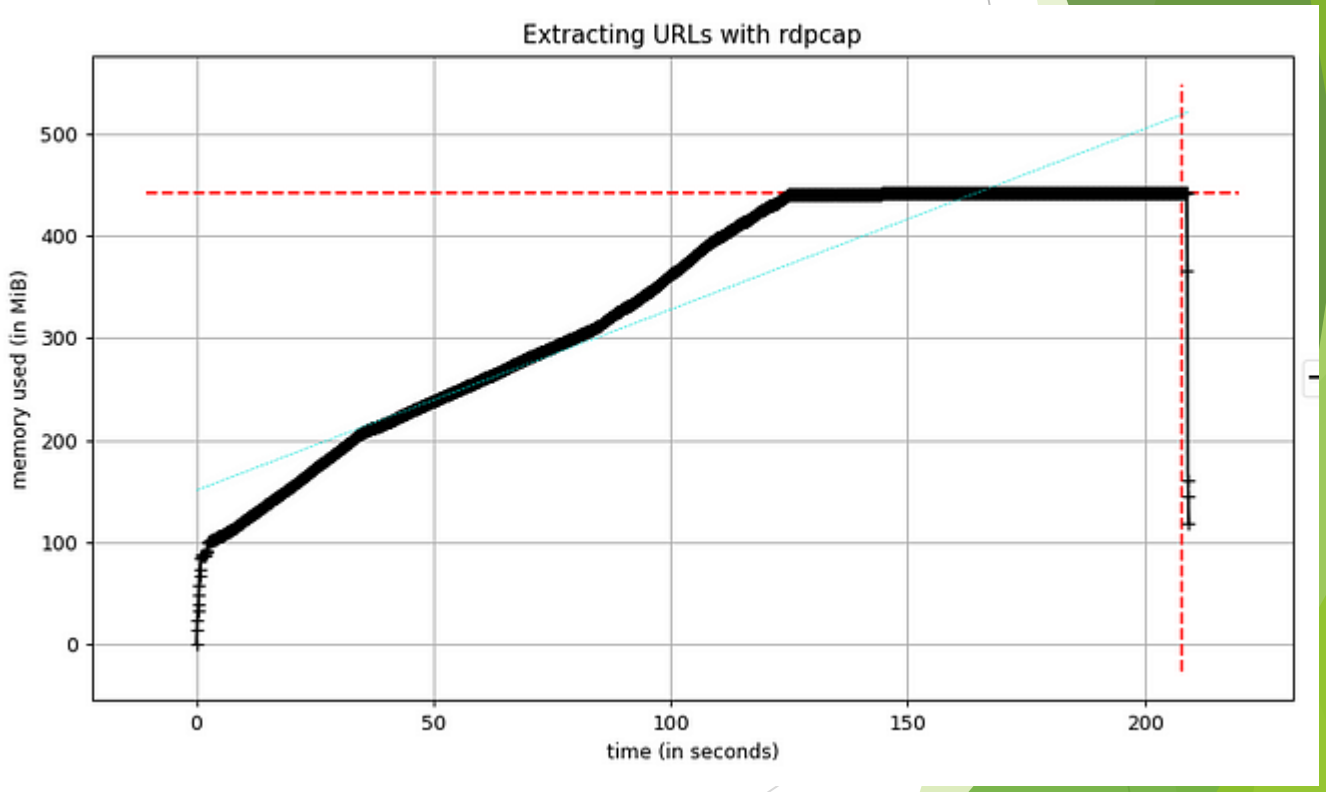
```
from memory_profiler import profile
import numpy as np.

mem_logs = open('mem_profile.log','a')

@profile(stream=mem_logs)
def memory_intensive_function():
    # Allocate a large list
    large_list = [i**2 for i in range(1000000)]

    # Allocate a large NumPy array
    large_array = np.ones((1000, 1000))

    return sum(large_list) + np.sum(large_array)

if __name__ == "__main__":
    memory_intensive_function()
```

Create a log file

mprof

- You can track memory usage over time using mprof.

1) Create a time profile:
   `mprof run script.py`
2) Plot the memory usage graph:
   `mprof plot`



Py-Spy

py-spy is a sampling profiler for Python that helps diagnose performance issues without modifying the code. It can attach to a running Python process, record function call statistics, and generate flame graphs for visualization.

- Attach to running processes (no restart needed)
- Low overhead (doesn't slow down your code)
- Works with multi-threaded Python programs
- Generates flame graphs for easy performance analysis
- Cross-platform (Linux, macOS, Windows)

Py-Spy

How to generat a profile:

1. Find your python proces
   ```
   ps aux | grep python
   ```
2. Generate a flame graph
   ```
   py-spy record -o profile.svg --pid <PID>
   ```
   or profile a script directly
   ```
   py-spy record -o profile.svg -- python my_script.py
   ```

Monitore running script:

1. Attach Py-Spy to running proces
   ```
   py-spy top --pid <PID>
   ```
   or profile a script directly
   ```
   py-spy top -- python my_script.py
   ```

Py-Spy

- PLOTS

Scalene

Scalene is a fast, high-resolution CPU, memory, and GPU profiler for Python. It helps pinpoint performance bottlenecks and memory leaks, and it even identifies time spent in Python vs. native code (C, C++, NumPy, etc.).

- Line-by-line CPU & memory profiling (better than cProfile)
- Distinguishes Python vs. native execution
- Tracks memory usage & leaks
- Supports multi-threading & multiprocessing
- Profiles GPU usage (for CUDA-based programs)
- Asynchronous profiling (low overhead, works with Jupyter Notebooks)

scalene

- `scalene my_script.py`

- CPU Profiling
  `scalene --cpu-only my_script.py`
- GPU Profiling
  `scalene --gpu-only my_script.py`
- Memory Profiling
  `scalene --memory-only my_script.py`

```
                         CPU %| Python %| Memory (MB)| Time (s) | File
-------------------------------------------------------------------------------
main.py:10  slow_function  90%     100%        10.5        2.0      my_script.py
main.py:25  fast_function  10%       0%         0.1        0.1      my_script.py
```

- Visualization
  `scalene -html my_script.py`

For Jupyter Notebooks:

```
%pip install scalene
%load_ext scalene
%scalene my_script.py
```