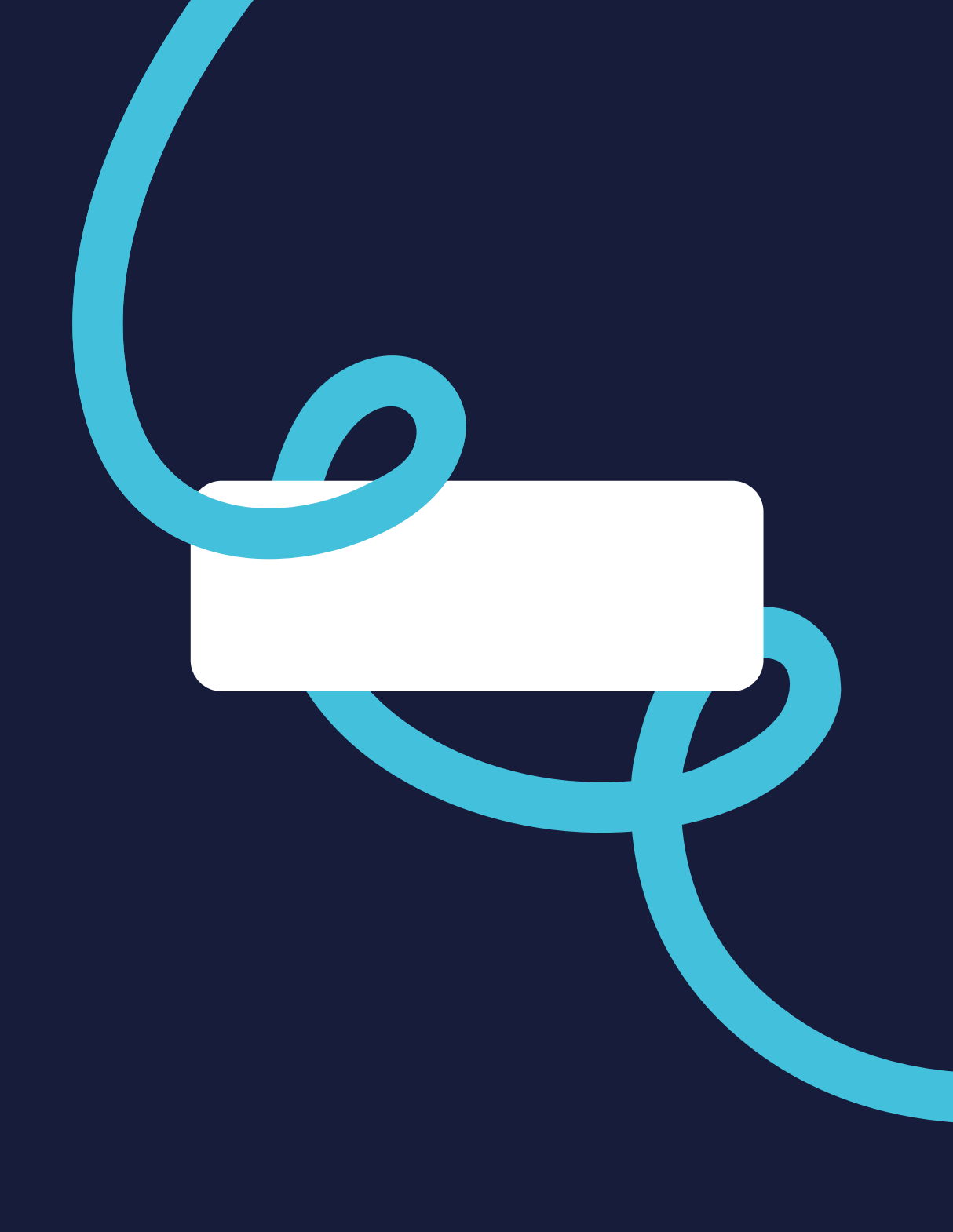

- Centific

04/16/24

Microsoft

Skills required:

- Native Python libraries
- Math: Sympy, Numpy, linalg
- Statistics library, pandas, sklearn
- scipy
- cvxpy : Convex optimization
- Object Oriented Development in Python.
- Ability to translate instructions that involve advanced math and physics concepts to code.

1. **Brush up on Python Fundamentals**
    - **Syntax and Constructs:** Be fluent in Python 3.9 features. Review comprehensions, decorators, generators, lambda functions, and the asynchronous features (asyncio).
    - **Object-Oriented Programming:** Understand classes, inheritance, polymorphism, and other OOP concepts. Practice writing clean, modular code using classes.
2. **Master the Required Libraries**
    - **Math and Science Libraries:**
    - **Numpy:** Be comfortable with array operations, broadcasting, and matrix arithmetic.
    - **Sympy:** Practice solving algebraic equations, integrating, differentiating, and other symbolic mathematics tasks.
    - **Scipy:** Get familiar with its capabilities in optimization, statistics, and signal processing.
    - **Data Handling and Statistics:**
    - **Pandas:** Know how to manipulate data frames, perform aggregations, and handle missing data.
    - **Sklearn:** Understand common machine learning algorithms and how to apply them using sklearn.
    - **Statistics:** Be able to perform statistical tests and data analysis using Python's statistics library.
    - **Optimization:**
    - **Cvxpy:** Learn how to model and solve optimization problems, especially those related to convex optimization.
3. **Practice Coding Problems**
    - **Platforms like LeetCode, HackerRank, and CodeSignal:** Focus on problems that require numerical calculations, statistics, or involve heavy data manipulation.
    - **Project-based Learning:** Consider creating small projects or scripts that utilize these libraries to solve complex problems or automate tasks.
4. **Understand the Domain of LLMs**
    - Since the job involves working with LLMs, gain a basic understanding of how these models work, especially in relation to Python coding.
    - Review how Python is used to interface with these models, likely through APIs or specific libraries that facilitate machine learning and AI tasks.
5. **Prepare for Behavioral Questions**
    - Be ready to discuss your past projects, particularly those involving Python and the libraries mentioned in the job description.
    - Think about times you've solved difficult problems, worked as part of a team, or had to learn something new quickly.
6. **Setup Your Interview Environment**
    - **Technical Setup:** Ensure your computer, webcam, and internet connection are stable. Set up a quiet, well-lit environment for the Skype interview.
    - **Coding Environment:** Have an IDE ready to go, preferably one you're familiar with, and possibly set up with the libraries you might need to use during the interview.
7. **Mock Interviews**
    - Consider scheduling mock interviews with friends or mentors in the tech field. Use platforms like Pramp or Interviewing.io, which offer free or paid mock technical interviews.
8. **Review the Google Python Style Guide**
    - Since the job description specifically mentions adherence to the Google Python Style Guide, make sure you are familiar with its conventions and best practices.

Preparing in these areas should give you a solid foundation to perform well in your live coding interview. Good luck!

Context Managers and Python Core

- Understand the purpose and function of context managers (`with` statement in Python).
- Know how to create a context manager using both the class-based method (`__enter__` and `__exit__`) and the `contextlib` module (using decorators).
- Review exception handling within context managers and ensure resources are properly cleaned up.
- Practice writing context managers for different resources like files, database connections, and network sessions.

Algorithm Design and Data Structures

- Practice problems that involve hash maps, sets, arrays, and other data structures.
- Understand common algorithms related to search, sort, and graph traversal.
- Familiarize yourself with modular arithmetic, which seems to have been a part of one of your interview questions.
- Review algorithms for string manipulation, especially if they involve complex string operations or pattern matching.

Pandas and Data Manipulation

- Gain proficiency in data manipulation using Pandas, including operations like `apply`, `merge`, `groupby`, `pivot_table`, and `explode`.
- Understand how to handle missing data and perform data cleaning.
- Learn how to perform data aggregation and summarization.
- Practice reading from and writing to different data formats (CSV, Excel, SQL databases).

Numpy and Scientific Computing

- Get comfortable with NumPy arrays and matrix operations, which are foundational for scientific computing in Python.
- Understand how to apply NumPy in solving mathematical problems, such as linear algebra or statistical analysis.

Advanced Python Features

- Deepen your understanding of list comprehensions, generator expressions, and iterators.
- Review decorators and how they can be used to enhance functions and methods.
- Understand asynchronous programming with `asyncio` if applicable to the role.

Problem Solving and Logic

- Enhance your ability to translate complex math and physics problems into code, which may involve utilizing libraries like `sympy` or `scipy`.
- Practice solving problems on a whiteboard or in an IDE to simulate interview conditions.

Soft Skills and Communication

- Practice explaining your code and thought process clearly and concisely.
- Prepare to discuss your previous projects, especially those relevant to the job description.
- Work on articulating how you approach problem-solving, how you overcome obstacles, and how you ensure your code is clean and maintainable.

** Let's start with Context Managers:

↳ Python Context Managers: used to manage resources.

↳ they are particularly useful because they allow you to work with resources in a clean and efficient way, making your code more readable and less error prone.

→ is an object that is used to manage resources, such as file handling, network connections, and DB connections.

↳ It provides a way to allocate and release resources automatically when they are no longer needed, ensuring proper cleanup and preventing resource leaks.

↳ They are typically used with the "with" statement, which guarantees that the resources managed by the context manager are properly released at the end of the block, even if exception occurs.

↳ Python provides two main ways to create context managers:

→ by defining a class with __enter__ and __exit__ methods,

→ by using contextlib module's contextmanager decorator.

Custom Context Manager:

Ex:-

```
f = open('sample.txt', 'w')
f.write('This is test.')
f.close()
```

↳ in this ex, we have to remember to close the file every time.

↳ so, it better to use context manager, which is helpful in case if we get any error, still it closes the file.

↳

```
with open('sample.txt', 'w') as f:
    f.write('This is test.')
```

- As we know, we can also use it to:
  - Connect and close the database connections automatically,
  - Acquire and release locks,
  - file handling,
  - Network connection, etc.

- To handle custom resources, we can write our own context managers.
  - ↳ We can either use a class or a function and decorator.

Ex: using Class:

```
Class Openfile:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
    def __enter__(self):
        self.file = open(self.filename,
                         self.mode)
        return self.file

    def __exit__(self, exc_type,
                 exc_val, traceback):
        self.file.close()

with Openfile('sample.txt', 'w') as f:
    f.write('Testing')

print(f.closed) # To check if file is closed
```

- We call the context manager - with, it comes to the __init__ method and sets the attribute.
- Then it runs the code within __enter__ method that opens the file and returns the variable - file.
- So, the f variable in the context manager, get the return variable

and we can work on this in any way we like to.

↳ Now, when we exit that block, it calls the __exit__ method that is calling self.file.close().

↳ So, when you print f.closed(), it will return True.

↳ exc_type, exc_val, traceback are the parameters to handle exceptions if desired by running 'True', though typically 'False' is returned to allow exceptions to propogate.

*→ Before getting into "how to use it through function and a decorator, a few revision points on both:

↳ Functions:
- first class objects which means that functions in Python can be used or passed as arguments.

Properties:
- it's an instance of the Object type.
- can store function in a variable
- can pass the func as a parameter to another func.
- can return the func from a func.

- can store them in data structure such as hash tables, lists, ---

Decorators:

- very powerful and useful tool in Python
- it allows programmers to modify the behavior of a func or class.
- allows us to wrap another func in order to extend the behavior of the wrapped func, without permanently modifying it.
- In Decorators, func are taken as the argument into another func and then called inside the wrapper func.

(*args, **kwargs) as arguments allows us to use any/unl number of arguments and key. arguments in a func.

Syntax:

Ex:-

```
@gfg_decorator
def hello_decorator():
    print("GFG")
```

↳ callable func
↳ it will add some code on the top of some another Callable func.

↓ is equivalent to :—

```
def hello_decorator():
    print("GFG")
hello_decorator = gfg_decorator(hello_decorator)
```

hello_decorator func and return the wrapper func

Back to how to use function and Decorator for context manager:

Ex:-

```
from contextlib import contextmanager

@contextmanager
def open_file(file, mode):
    f = open(file, mode)
    yield f
    f.close()

with open_file('sample.txt', 'w') as f:
    f.write('This is a test')

print(f.closed)
```

↳ Here the func open_file is doing the same thing which we did in class.

↳ We are here using yield statement.

↳ Everything before yield is same as the __enter__ method in class.

↳ yield is the code within the with statement is going to run, treat as it is same as when we try to return the file in the __enter__ method in class.

↳ Everything after the yield statement is same as the __exit__ method in class.

2nd

Let's revise OOP:

- First some basic def^n:—

Func^c: block of code that performs a specific task.
↳ can take zero or more arguments, can return one or more values.
↓ can be called by name

Class: is a blueprint for creating objects.
↳ defines a set of attributes and methods that characterize any object of the class.
↳ We use classes to encapsulate data and the operations that must be performed on that data.

Method: is a func^c that is defined inside a class body.
↳ if called on an instance of that class, the method is said to be an "instance method" and it operates on that data (attributes) of that instance.
↳ Methods provide a way to interact with and manipulate the object data.

self: a parameter in Python that refers to the instance of the class.
↳ automatically passed with a method call from an object allowing the method to access the attributes and other methods of the object. It's similar to 'this' in other programming lang.

Feature	Function	Method
Definition	A block of code that performs a specific task.	A block of code that is associated with an object or class.
Arguments	Can take zero or more arguments.	Can take zero or more arguments, but the first argument must be the object on which the method is being called.
Return value	Can return one or more values.	Can return one or more values, but the first return value is typically the object on which the method was called.
Calling	Called by name.	Called on an object using dot notation.

Functions are independent blocks of code that can be called from anywhere, while methods are tied to objects or classes and need an object or class instance to be invoked.

* __init__ : special method in Python class.
  - Constructor method for a class.
  - called automatically when a new object of a class is instantiated.
  - This method is typically used to initialize the attributes of the new object.

* Argument : is a value that is passed to a func. (or method) when it is called.
  - By using arguments, func can perform operations on variable data.
  - func can take any no. of arguments.

* Tuple :- is a collection which is ordered and immutable (unchangeable).
  - are written with round brackets and can hold a mix of different data types.
  - often used to store related pieces of information that are not meant to be modified, such as the coordinates of a point.

* Instance : refers to specific object created from a particular class.
  - When a class is defined, no memory is allocated until an object is

created from a class, which is an instance.

↳ An instance embodies all the properties and behaviors specified by the class, but with specific values that differentiate it from other instances of the same class.

↳ They are fundamental in OOP because they allow programmers to create numerous objects, each with potentially different initial projects, to behave in ways defined by their class.

* Object :- entity that has state and behavior.
  ↳ is an instance of a class, where the class defines the blueprint for the object.
  ↳ It encapsulates data and the operations that can be performed on that data.
  ↳ They interact with each other through methods, the specific implementation of behaviors.
  ↳ They hold data in attributes and exhibit behavior through methods.

** Destructor: a special method that is called when an object is about to be destroyed. It is often used for cleanup activities, such as closing files or releasing resources.
↳ Not commonly used or necessary in Python due to its garbage collection system.

** Inheritance: A mechanism by which a new class can inherit the attributes and methods of an existing class.
↳ new class → called derived (or child) class.
↓
which inherits from base (or parent) class.

** Polymorphism: ability of different objects to respond, each in its own way, to the same message (or method call).
↳ In practice, this means that a method can be implemented differently across multiple classes.

* Encapsulation: Concept of bundling the data (attributes) and methods that operates on the data into a single unit and restricting access to some of objects components.
  - This is usually done by making some attributes or methods private, which means they can't be accessed from outside the class.

* Abstraction: A concept that involves hiding complex realities while exposing only the necessary parts of objects.
  - helps in reducing programming complexity and efforts.

* Interface: An abstract type that contains no data but defines behaviors as method signatures.
  - Classes that implement the interface agrees to implement all the methods defined by the interface.

* Access modifiers: Keywords that set the accessibility of classes, methods, and other members.

↳ Common modifiers include 'public', 'private', and 'protected', controlling how and where the class members can be accessed.

** Overloading: The ability to define several methods all with the same name but different parameters.
↳ Python does not support method overloading, but it can be simulated or worked around.

** Overriding: A feature that allows a subclass to provide a specific implementation of a method that is already defined in its parent class.

** yield: keyword that is used in a fun() to make it a generator.
↳ A generator is a special type of iterable that generates values on the fly, which means it does not store all of its values in memory at once. Instead, it yields one value at a time, which allows for better memory efficiency, especially when dealing with large datasets.

lambda:

fun^c is a small anonymous fun^c defined using 'lambda' keyword.

↳ Unlike a normal fun^c defined with 'def', a lambda fun^c is a single expression that returns a value (without using a return statement).

↳ It is typically used for creating small, one-off fun^c that are not complex enough to warrant a full fun^c definition using 'def'.

Syntax:

lambda arguments : expression

- arguments → refers to list of params that fun^c takes (i/p to arguments in regular fun^c)
- expression → single expression that gets evaluated and returned when the lambda is called

Simple Ex:—

```
add_ten = lambda x : x+10
print(add_ten(5))
```

o/p:→ 15

try and finally: Parts of exception handling that allow developers to manage errors gracefully and ensure that certain cleanup actions are executed, regardless of whether an error occured in the 'try' block.

'try' block lets you test a block of code of errors, while the 'finally' block lets you execute code, regardless of the result of the try-and except blocks.

Basic structure:

```
try:
    # Code that might throw an
    # exception
    risky_operation()
except Exception as e:
    # code that runs if an
    # exception occurs
    handle_exception(e)
finally:
    # code that runs no matter
    # what
    cleanup_action()
```

↳ if you open a file in the 'try' block, you should close it in the 'finally' block to ensure the file is closed whether or not an exception occurs.

↳ Similarly, other resources like network connections or database connections should be released in the 'finally' block.

↳ Restoring state: if your 'try' block alters an important part of the application state, you might use 'finally' to ensure that the state is reset appropriately.

** Asynchronous features:

↳ facilitated by the 'asyncio' library, allow parts of your program to run concurrently.

↳ 'async' and 'await' are keywords used to define and handle asynchronous operations.

** async: keyword is used to declare a func as an asynchronous func.

↳ Such func are called "coroutines" and can be paused and resumed at many points (unlike regular func that

run from start to finish).

↳ 'await': The 'await' keyword is used inside an async func to suspend the execution of the coroutine until the result of an awaitable object (like another coroutine) is returned.

↳ Event loop: core feature of 'asyncio' is the event loop, which is the central execution device provided by the library.
↳ It handles and manages the execution of several tasks concurrently by maintaining a queue of tasks to execute, executing them, and handling the I/O events.

↳ Coroutines: defined using 'async def'
↳ Special func that, unlike regular func, can suspend their execution before reaching return and can indirectly pass control back to the event loop.

↳ Tasks: used to schedule coroutines concurrently.
↳ when a coroutine is wrapped into a task with func like 'asyncio.create_task()', the event loop can take care of managing its execution.

Futures: objects that represent the result of work that has not yet completed.
↳ they are used to synchronize program execution at a high level.

Asynchronous I/O, Event loop, and Protocols:

- 'asyncio' provides a set of high-level APIs for:
  - Network programming.
  - Running subprocess
  - Handling OS signals.
  - Performing high-level file operations.

Ex:→

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(),
                         count(),
                         count())
if __name__ == "__main__":
    import time
    start = time.perf_counter()
```

```
asyncio.run(main())
elapsed = time.perf_counter() - start
print(f"Executed in {elapsed:0.2f} seconds.")
```

↳ asyncio.sleep(): simulates an I/O-bound operation by sleeping.

↳ asyncio.gather(): used to run multiple coroutines concurrently as tasks.

↳ asyncio.run(): main entry point for asynchronous programs and is used to execute the main coroutine along with all other coroutines it awaits.

⇒ Diff b/w '.format()' and 'f-strings':

- '.format()' and 'f-strings' are both used for string formatting in Python,
  - ↳ with .format() being more suitable for situations where the template needs to remain constant.
  - ↳ 'f-strings' offer more readability and performance benefits in most other cases.

map()

return iterators that can be converted into lists or tuples, can be iterated over

: used to apply a specific func to each item of an iterable (like a list) and return an iterator that yields the results.

Syntax: map(func, iterable, ...)

filter()

performs a repetitive operation over the pairs of the iterable. The result of the operation on the first two elements is used along with the next element and this continues until no more elements are left.

: used to create an iterator from elements of an iterable for which a func returns true.

↳ in other words, it filters the given iterable with the help of a func that tests each element in the iterable to be true or not.

Syntax: filter(func, iterable)

reduce()

Ex: summing all items

Returns a single cumulative value

: used to apply a particular func passed in its argument to all of the list elements mentioned in the sequence passed along.

↳ This func is available in 'functools' module and is used for consolatively performing an operation on all the items.

Syntax: reduce(func, iterable, [initializer])

** Naming Conventions :

- **Modules** : names should be short, lowercase and underscore can be used.
  Ex : my_module.
- **Classes** : use the CapWords convention
  Ex : MyClass
- **Func and variables** : func name should be lowercase, with words separated by underscores
  Ex : my_function.
- **Constants** : usually defined on a module level and written in all capital letters with underscores separating words.
  Ex : MY_CONSTANT

- **Code layout** :
  - **Indentation** : use 4 spaces per indentation level, never tabs.
  - **Line length** : limit most lines to 79 characters; docstrings and comments to 72 characters.
  - **Blank line** : use to separate func and classes, and larger blocks of code inside func.

↳ imports: on top of file.
  ↳ on separate lines.
  ↳ placed b/w (module, comments, docstrings) and (module globals and constraints)

↳ Whitespaces in Expressions and Statements:

⇒ Avoid extraneous whitespace in the following situations:
  ↳ Immediately inside parenthesis, brackets, or braces.
  ↳ b/w a trailing comma and a following close parenthesis.
  ↳ Immediately before a comma, semicolon or colon.

↳ Use spaces around arithmetic operators.

⇒ Comments: → should be complete sentences.
→ if a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower-case letters.

↳ Inline comments: Use inline comments sparingly and avoid obvious comments.

↳ Docstrings: → write this for all public modules, functions, classes, and methods.

↳ should start with a capital letter and end with a period.

↳ usage: triple-double quote (""")

↳ Python is Case-Sensitive.