

Assignment-9.4

Name: Arjun Manoj

H. No:2303A52134

Batch:44

Task 1: Auto-Generating Function Documentation in a Shared Codebase

Ask the AI to automatically generate Google-style function docstrings for each function

- Each docstring should include:
- A brief description of the function
- Parameters with data types
- Return values
- At least one example usage

Prompt:

"Generate Google-style docstrings for the following Python function."

Code:

The screenshot shows the VS Code editor with the file 'Assignment-9.4.py' open. The code defines a function `classify_number(n)` with a docstring that includes a description, parameters, return values, and examples. The docstring is as follows:

```
1 # Automatically generate Google-style function docstrings for each function
2 # * Each docstring should include:
3 # * A brief description of the function
4 # * Parameters with data types
5 # * Return values
6 # * At least one example usage (if applicable)
7 # Experiment with different prompting styles (zero-shot or context-based)
8 # to observe quality differences.
9 def classify_number(n):
10     """Classifies a number as Positive, Negative, or Zero.
11
12     Args:
13         n (int or float): The number to be classified. Can also be None or a string for invalid input.
14
15     Returns:
16         str: A string indicating whether the number is "Positive", "Negative", "Zero", or "Invalid input".
17
18     Example:
19     >>> classify_number(10)
20     'Positive'
21     >>> classify_number(-5)
22     'Negative'
23     >>> classify_number(0)
24     'Zero'
25     >>> classify_number("invalid")
26     'Invalid input'"""
```

The terminal at the bottom shows the command `python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-9.4.py"` being executed, with the output:

```
Positive
Negative
Zero
Invalid input
Invalid input
Invalid input
```

The screenshot shows the VS Code editor with the file 'Assignment-9.4.py' open. The code defines a function `classify_number(n)` with a docstring and test cases. The code is as follows:

```
9 def classify_number(n):
26     'Invalid input'
27     >>> classify_number(None)
28     'Invalid input'
29
30     if n is None or isinstance(n, str):
31         return "Invalid input"
32     elif n > 0:
33         return "Positive"
34     elif n < 0:
35         return "Negative"
36     else:
37         return "Zero"
38
39 # Assert test cases
40 assert classify_number(10) == "Positive" # Test with positive number
41 assert classify_number(-5) == "Negative" # Test with negative number
42 assert classify_number(0) == "Zero" # Test with zero
43 assert classify_number("invalid") == "Invalid input" # Test with string input
44 assert classify_number(None) == "Invalid input" # Test with None input
45 print(classify_number(10))
46 print(classify_number(-5))
47 print(classify_number(0))
48 print(classify_number("invalid"))
49 print(classify_number(None))
50
```

The terminal at the bottom shows the command `python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-9.4.py"` being executed, with the output:

```
Positive
Negative
Zero
Invalid input
Invalid input
Invalid input
```

Observation:

- The AI generated a basic docstring.
- Description and parameters were included.
- Examples were sometimes missing or minimal.
- Formatting was correct but less detailed.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

You are provided with a Python script containing:

- Loops
- Conditional logic
- Algorithms (such as Fibonacci sequence, sorting, or searching)

Use AI assistance to:

- Automatically insert inline comments only for complex or non-obvious logic
- Avoid commenting on trivial or self-explanatory syntax

The goal is to improve clarity without cluttering the code.

Prompt:

Insert meaningful inline comments in the following Python code.
Comment only complex or non-obvious logic and avoid explaining basic Python syntax

Code:

Observation

- The AI added comments only where the logic was not immediately obvious.
- The tuple update in the Fibonacci algorithm was clearly explained.
- Trivial lines like variable initialization and loop syntax were not commented.
- Code readability improved without adding unnecessary clutter.

Task 3: Generating Module-Level Documentation for a Python Package

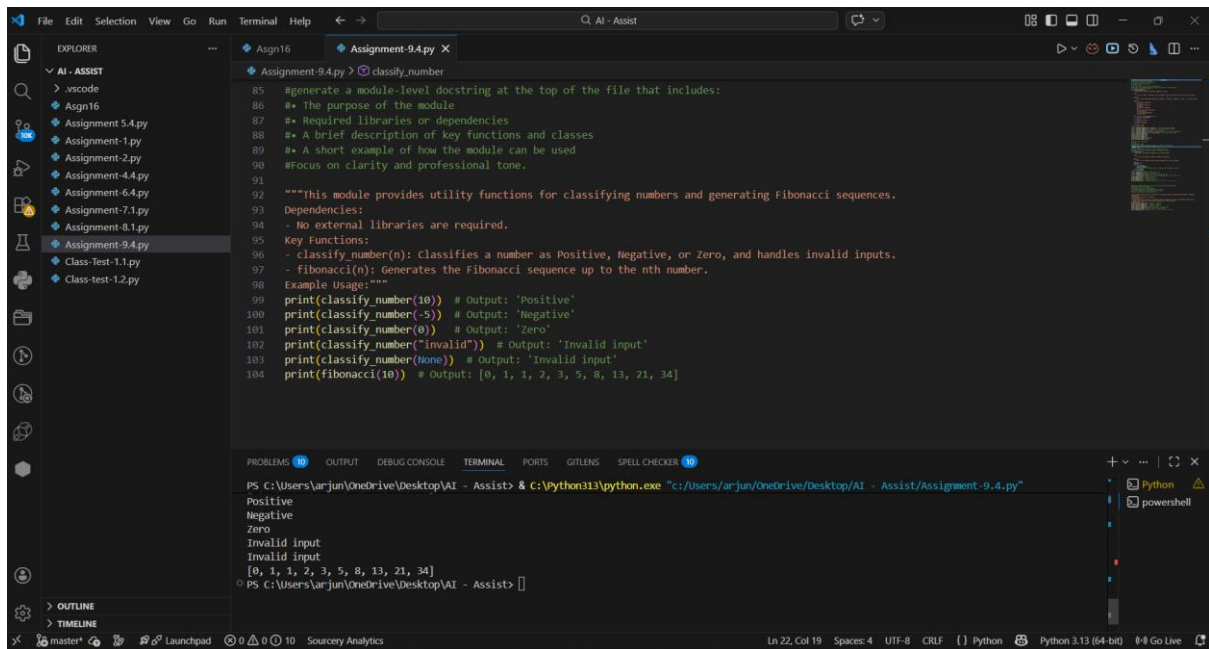
Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:

- The purpose of the module
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example of how the module can be used

Prompt:

Generate a professional module-level docstring for this Python module. Include the purpose, dependencies, key functions, and example usage

Code:



Observation

- The AI generated a structured multi-line docstring at the top of the file.
- The documentation clearly explained the module's purpose and functionality.
- Dependencies and key functions were summarized concisely.
- Example usage improved clarity for new users of the module.
- The tone was professional and suitable for real-world repositories.

Task 4: Converting Developer Comments into Structured Docstrings

You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

- Automatically convert these comments into structured Google-

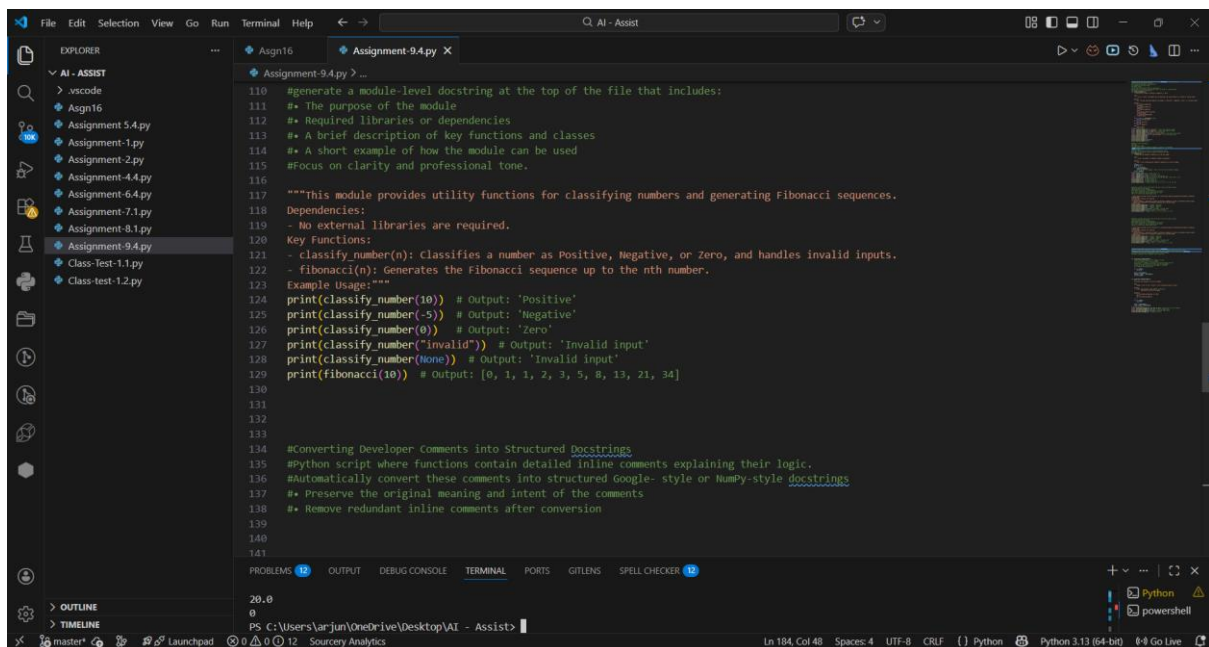
style or NumPy-style docstrings

- Preserve the original meaning and intent of the comments
- Remove redundant inline comments after conversion

Prompt:

Convert the following inline explanatory comments into a structured Google-style docstring. Preserve the original meaning and remove redundant inline comments

Code:



```
110 #generate a module-level docstring at the top of the file that includes:
111 #* The purpose of the module
112 #* Required libraries or dependencies
113 #* A brief description of key functions and classes
114 #* A short example of how the module can be used
115 #focus on clarity and professional tone.
116
117 """This module provides utility functions for classifying numbers and generating Fibonacci sequences.
118 Dependencies:
119 - No external libraries are required.
120 Key Functions:
121 - classify_number(n): Classifies a number as Positive, Negative, or Zero, and handles invalid inputs.
122 - fibonacci(n): Generates the Fibonacci sequence up to the nth number.
123 Example Usage: """
124 print(classify_number(10)) # Output: 'Positive'
125 print(classify_number(-5)) # Output: 'Negative'
126 print(classify_number(0)) # Output: 'Zero'
127 print(classify_number("invalid")) # Output: 'Invalid input'
128 print(classify_number(None)) # Output: 'Invalid input'
129 print(fibonacci(10)) # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
130
131
132
133
134 #Converting Developer Comments into Structured Docstrings
135 #Python script where functions contain detailed inline comments explaining their logic.
136 #Automatically convert these comments into structured Google- style or NumPy-style docstrings
137 #* Preserve the original meaning and intent of the comments
138 #* Remove redundant inline comments after conversion
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
142 def calculate_average(numbers):
143     # This function takes a list of numbers as input.
144     # It first checks if the list is empty.
145     # If the list is empty, it returns 0 to avoid division by zero.
146     # Otherwise, it calculates the total sum of the numbers.
147     # Then it divides the total by the number of elements
148     # to compute the average value.
149
150     if not numbers:
151         return 0
152
153     total = sum(numbers)
154     average = total / len(numbers)
155     return average
156
157
158 def calculate_average(numbers):
159     """
160     Calculates the average of a list of numbers.
161
162     Args:
163         numbers (list of int or float): A list containing numeric values.
164
165     Returns:
166         float: The average of the numbers in the list.
167         Returns 0 if the list is empty.
168
169     Examples:
170         >>> calculate_average([10, 20, 30])
171         20.0
172         >>> calculate_average([])
173         0
174
175     """
176     if not numbers:
177         return 0
178
179     total = sum(numbers)
180     return total / len(numbers)
181
182 # Assert test cases
183 assert calculate_average([10, 20, 30]) == 20.0 # Test with a list of numbers
184 assert calculate_average([]) == 0 # Test with an empty list
185 print(calculate_average([10, 20, 30])) # Should return 20.0
186 print(calculate_average([])) # Should return 0
```

```
158 def calculate_average(numbers):
159     """
160     Calculates the average of a list of numbers.
161
162     Args:
163         numbers (list of int or float): A list containing numeric values.
164
165     Returns:
166         float: The average of the numbers in the list.
167         Returns 0 if the list is empty.
168
169     Examples:
170         >>> calculate_average([10, 20, 30])
171         20.0
172         >>> calculate_average([])
173         0
174
175     """
176     if not numbers:
177         return 0
178
179     total = sum(numbers)
180     return total / len(numbers)
181
182 # Assert test cases
183 assert calculate_average([10, 20, 30]) == 20.0 # Test with a list of numbers
184 assert calculate_average([]) == 0 # Test with an empty list
185 print(calculate_average([10, 20, 30])) # Should return 20.0
186 print(calculate_average([])) # Should return 0
```

Observation:

- The AI successfully converted descriptive inline comments into a structured docstring.
- The original meaning of the logic was preserved.
- Redundant inline comments were removed.
- The function body became cleaner and more readable.

- Documentation format became consistent with professional standards.

Task 5: Building a Mini Automatic Documentation Generator

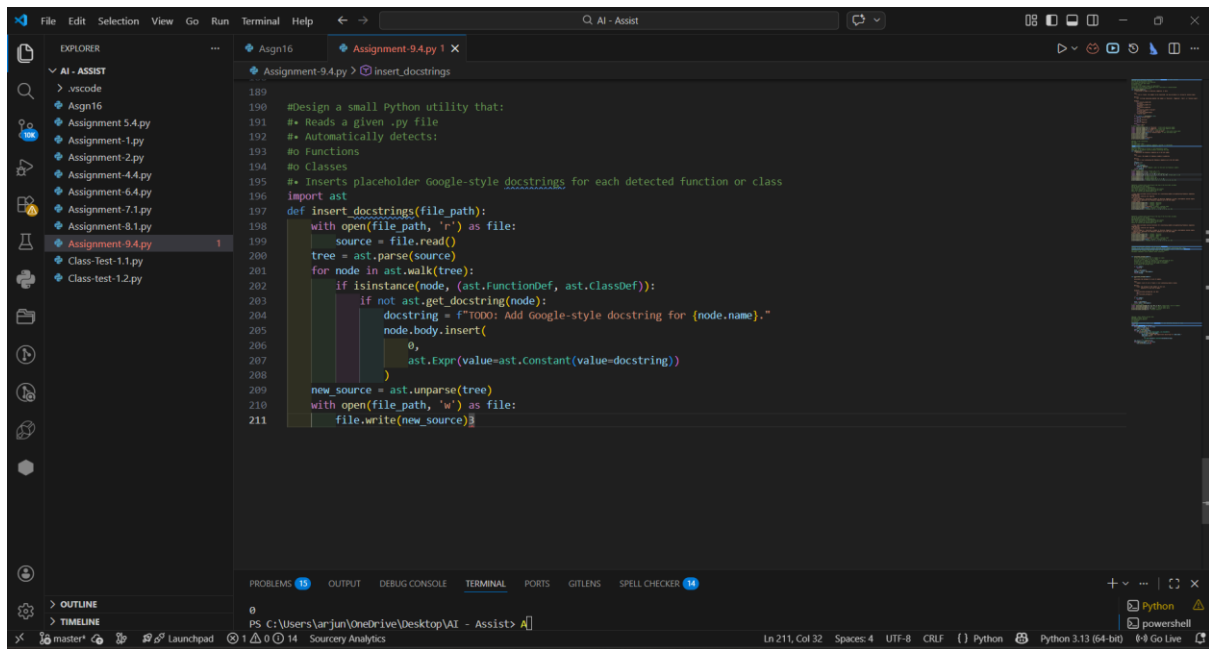
Design a small Python utility that:

- Reads a given .py file
- Automatically detects:
 - Functions
 - Classes
- Inserts placeholder Google-style docstrings for each detected function or class

Prompt:

Help me build a Python utility that reads a .py file, detects functions and classes using AST, and inserts placeholder Google-style docstrings where missing

Code:



Observation:

- AI suggested using the ast module to parse and modify the syntax tree.
- Functions and classes were automatically detected.
- Placeholder docstrings were inserted only where missing.
- The tool successfully scaffolded documentation without affecting existing docstrings.
- Demonstrates automation of documentation setup.