CoGrammar

Week 9 – Open Class 2

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Software Engineering Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
  (FBV: Mutual Respect.)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes. You can submit these questions here: **Open Class Questions**

Software Engineering Lecture Housekeeping cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**

Progression Criteria

✓ Criterion 1: Initial Requirements

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

✓ Criterion 2: Mid-Course Progress

- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

✓ Criterion 3: Post-Course Progress

- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.

✓ Criterion 4: Employability

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.

Lecture Objectives

1. Review Best Practices related to functions covered in the previous session.
2. Best Practices examples.
3. Open floor: Q&A

Calling Functions

- ★ Functions with one required positional input:
  - my_function(input1)
- ★ Functions with two required positional inputs:
  - my_function(input1, input2)
- ★ Functions with one required positional input and one optional keyword input:
  - my_function(input1, keyword_arg=input2)
- ★ Functions with variable positional inputs/keyword inputs:
  - my_function(*args, **kwargs)

Defining our own Functions

★ Uses the def keyword (for define):

- def add_one(x): # function called add_one

```
    y = x + 1
    return y
```

★ Important keywords:

- def - tells Python you are defining a function
- return - if your function returns a value, then use this keyword to return it.

Why Functions?

- ★ **Reusable code** - Sometimes you need to do the same task over and over again.
- ★ **Error checking/validation** - Makes this easier, as you can define all rules in one place.
- ★ **Divide code up into manageable chunks** - Makes code easier to understand.
- ★ **More rapid application development** - The same functionality doesn't need to be defined again.
- ★ **Easier maintenance** - Code only needs to be changed in one place.

★ **Building Custom decorators:**
Here we're crafting a decorator that measures a functions execution time.

```python
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time} seconds to run.")
        return result
    return wrapper

@timing_decorator
def slow_function():
    # Some time-consuming task
    pass
```

★ **Type Annotations :**

Type hints make your code more understandable by allowing developers to see what types of arguments a function expects while giving them an idea of what the function will return:

```python
def add_numbers(a: int, b: int) -> int:
    return a + b

# If you want to indicate types like List or Dict:
from typing import List

def process_data(data: List[str]) -> None:
    # Code to process the data
    data.sort()
```

★ **Return Types:**

Try to keep return types consistent. This will remove the issue of having trouble anticipating the return type for the function.

```
def find_item(item, item_list):
    if item in item_list:
        return True
    else:
        return "Item not in list"

# The better alternative:
def find_item(item, item_list):
    return item in item_list
```

Wrapping Up

Best Practices

In conclusion, embracing best practices in your Python functions not only enhances the clarity and maintainability of your code but also elevates you as a conscientious and effective developer.

Docstrings

In summary, crafting informative docstrings and integrating Sphinx for documentation elevates your code into a comprehensive and accessible resource that fosters collaboration & understanding among developers.

Wrapping Up

Type Annotations

To sum it up, embracing type annotations not only improves the clarity and predictability of your Python functions but lays the groundwork for a robust and maintainable codebase, ultimately improving its scalability.

Returns

In a nutshell, adopting consistent return practices not only streamline your code but fosters a more predictable and graceful interaction between functions.

CoGrammar

Thank you for joining