Things Learned While Applying for Jobs

jueves, 23 de noviembre de 2023

1:03 p. m.

* During the process in Turing I stumbled upon questions I know I can solve easily, so I'm addressing them one by one.

Things I can solve now…

* ~~Do private attribute can be accessed by derived classes?~~
* ~~What's the correct order or arguments in functions?~~
* ~~The filter() function.~~

Do private attribute can be accessed by derived classes?

If is a matter of capability, derived classes are capable of accessing private class or instance attributes, the thing is that by adding the preceding dunder makes it a little more difficult to directly access them.

' With name-tricking it'

class A:

    \_\_priv\_a\_cls\_att = 'Private A class Attribute'

    def \_\_init\_\_(self):

        self.\_\_priv\_a\_ins\_att = 'Private A class Instance Attribute'

        pass

class B(A):

    pass

print(f"Accessing from an instance of A: {A().\_A\_\_priv\_a\_ins\_att}") # Accessing from an instance of A: Private A class Instance Attribute

print(f"Accessing from the A Class: {A.\_A\_\_priv\_a\_cls\_att}", end="\n\n")    # Accessing from the A Class: Private A class Attribute

print(f"Accessing from an instance of B: {B().\_A\_\_priv\_a\_ins\_att}") # Accessing from an instance of B: Private A class Instance Attribute

print(f"Accessing from the B Class: {B.\_A\_\_priv\_a\_cls\_att}")    # Accessing from the B Class: Private A class Attribute

'With ChatGPTs Example'

class Parent:

    def \_\_init\_\_(self):

        self.\_\_private\_attr = 10

    def get\_private\_attr(self):

        return self.\_\_private\_attr

class Child(Parent):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def access\_private\_attr(self):

        # This won't work directly:

        # print(self.\_\_private\_attr)

        # But this will, using the method from the parent class:

        print(self.get\_private\_attr())

child = Child()

child.access\_private\_attr()  # This will print 10

What's the correct order or arguments in functions?

The correct order is:

1. Positional.
2. Default (Or optional).
3. Arbitrary positional (\*args).
4. Keyword.
5. Keyword default.
6. Arbitrary Keyword (\*\*kwargs).

# This one

def example\_function(pos\_arg1, pos\_arg2, default\_arg1="default", \*args, kwarg1, kwarg2="value", \*\*kwargs):

    # pos\_arg1 and pos\_arg2 are positional arguments

    # default\_arg1 is a default argument

    # \*args collects extra positional arguments

    # kwarg1 and kwarg2 are keyword arguments with kwarg2 having a default value

    # \*\*kwargs collects extra keyword arguments

    print("Positional arguments:", pos\_arg1, pos\_arg2)

    print(f"Default argument: {default\_arg1} / which is still needed to be passed")

    print("Arbitrary positional arguments (\*args):", args)

    print(f"Keyword arguments:'{kwarg1}', '{kwarg2}'")

    print("Arbitrary keyword arguments (\*\*kwargs):", kwargs)

# Example function call

example\_function(1, 2, "default", 'arb\_arg1', 'arb\_arg2', 'arb\_arg3', kwarg1="keyword\_arg", extra\_arb\_kwarg1="kw value 1", extra\_arb\_kwarg2="kw value 2")

The output will be:

Positional arguments: 1 2

Default argument: default / which is still needed to be passed

Arbitrary positional arguments (\*args): ('arb\_arg1', 'arb\_arg2', 'arb\_arg3')

Keyword arguments:'keyword\_arg', 'value'

Arbitrary keyword arguments (\*\*kwargs): {'extra\_arb\_kwarg1': 'kw value 1', 'extra\_arb\_kwarg2': 'kw value 2'}

The filter() function?

The filter() function in Python is used to filter elements from an iterable (like a list) based on a specified function. It returns an iterator that contains the elements for which the function returns True.

After comparing with a range of 10.000 numbers to filter the even numbers with: filter func with a call to a separate function, a filter func with a lambda func and a listcomp, the listcomp end up being always more efficient and pythonic. the other two are head to head in runtime performance.