Overview

This assignment is to test a few key attributes we are looking for in a new team member. Most of these questions are not realistic, however, we hope that they serve as a way for you to demonstrate how you work and organise yourself. The questions are intentionally brief and simple, since we know that this is not the only job opportunity you are exploring.

Performance

Suppose the `get_resource_identifier` function interrogates some cloud infrastructure to resolve the resource identifier from its name. Note that it takes a long time for the call to finish resolving the name.

Now imagine that we need to resolve the resource by its name multiple times during deployment of infrastructure. How can we speed this up without modifying the body of the `get_resource_identifier` function? Remember, you have no control over how quickly the cloud provider can respond to your API call.

In [ ]:

```python
import time
def get_resource_identifier(name):
    time.sleep(1)#simulate the delay
    if name is 'foo':
        return 'L9UKvnomjq'
    if name is 'bar':
        return '7U9eyOv7M'
    return 'Not found'

"""
for _ in range(0,100):
    print(get_resource_identifier('foo'))
    print(get_resource_identifier('bar'))
    print(get_resource_identifier('foo'))
    print(get_resource_identifier('zoo'))
    print(get_resource_identifier('bar'))
"""
# Solution for performance:

server_responses = dict(foo=get_resource_identifier('foo'),
                        bar=get_resource_identifier('bar'),
                        zoo=get_resource_identifier('zoo'))

for _ in range(0,100):
    print(server_responses['foo'])
    print(server_responses['bar'])
    print(server_responses['foo'])
    print(server_responses['zoo'])
    print(server_responses['bar'])
```

Readability and simplicity

Refactor

The section below is an opportunity for you to demonstrate how you refactor code into something simpler and more readable. Refactor the code and write some very simple sanity checks to show that the refactored version is equivalent to the ugly version. You may leave out tests where you think it is not needed.

In [ ]:

```python
# Don't modify this
colours = ['blue','green','yellow','black','orange']
fruits = ['berry','apple','banana','currant']
# All of the rest below you may modify
# as you please to achieve the desired output
```

In [ ]:

```
#ugly
ugly_output = [] #sanity check
better_output = [] #sanity check

for i in range(len(colours)-1,-1,-1):
    print(colours[i])
    ugly_output.append(colours[i])

#refactor below

print()
colours_reversed = list(reversed(colours))
for colour in colours_reversed:
    better_output.append(colour)
    print(colour)

print(f"Are the outputs the same: {ugly_output == better_output}")
```

In [ ]:

```
#ugly
ugly_output = [] #sanity check
better_output = [] #sanity check

for i in range(len(colours)):
    ugly_output.append({i,colours[i]})
    print(i,colours[i])

#refactor below

print()

colours_enumerated = enumerate(colours)
for index,colour in colours_enumerated:
    better_output.append({index,colour})
    print(index,colour)

print(f"Are the outputs the same: {ugly_output == better_output}")
```

In [ ]:

```
#ugly
ugly_output = [] #sanity check
better_output = [] #sanity check

min_length = min(len(colours),len(fruits))
for i in range(min_length):
    ugly_output.append({colours[i],fruits[i]})
    print(colours[i],fruits[i])

#refactor below

print()

for colour,fruit in zip(colours,fruits):
    better_output.append({colour,fruit})
    print(colour,fruit)

print(f"Are the outputs the same: {ugly_output == better_output}")
```

In [ ]:

```
#ugly
#you may deal with these variables in the abstract
#you can give them values if you want to do some
#sanity checks

a = 1
b = 2
c = 3
d = 4
f = 5
g = 6
```

```
if a <= b and f <= g and c<=d and d<=f and b<=c:
    print('pass')
else:
    print('fail')

#refactor below

# If rearranged:
# a <= b and b <= c and c <= d and d <= f and f <= g
# this means all we have to do is check if the elements are in
# acsending order to get the same result since each previous element is smaller than the last.

#abstract_variables = [a,b,c,d,f,g]
abstract_variables = [1,2,3,4,5,6]
if abstract_variables == sorted(abstract_variables):
    print('pass')
else:
    print('fail')
```

Implement

This section provides an opportunity to demonstrate how you would write some very simple things in a pythonic way.

Task1:

In [ ]:

```
#Generate the following string from the colours list defined above:
# 'blue --> green --> yellow --> black --> orange'

output_string = ' --> '.join(colours)
print(output_string)
```

Task2

In [ ]:

```
# find the elements that exist in the first list but not the second
# and the elements that exist in the second, but not in the first
# put this result in into a single list and sort them in ascending order


first = [2,2,5,6,7,2,1,8,9,9]
second = [2,1,5,6,66,7,77]

result = []

[result.append(number) for number in first if number not in second]

[result.append(number) for number in second if number not in first]

result.sort()
print(result)
```

In [ ]:

```

```