PYTHON IN CLOUD (ANGOL NYELVŰ)

1. forduló

CAMBRIDGE MOBILE TELEMATICS

A kategória támogatója: Cambridge Mobile Telematics (CMT)

Ismertető a feladathoz

Kérjük, hogy a feladatlap indítása előtt mindenképp olvasd el az alábbi útmutatót:

- MINDEN kérdésre **van helyes válasz**.
- Olyan kérdés **NINCS**, amire az összes válasz helyes, ha mégis az összes választ bejelölöd, arra a feladatra automatikusan 0 pont jár.
- A **radio button-os kérdésekre egy helyes válasz van**.
- **Ha lejár a feladatlap ideje, a rendszer AUTOMATIKUSAN** beküldi azt az addig megjelölt válaszokkal.
- Azokat a feladatlapokat, amelyekhez **csatolmány** tartozik, javasoljuk **NEM mobilon** elindítani, erre az érintett feladatlapok előtt külön felhívjuk a figyelmet.
- Az **adatbekérős feladatokra NEM jár részpontszám**, csak a feleletválasztósakra.
- **Helyezéseket a 4. forduló után mutatunk**, százalékos formában: adott kategóriában a TOP 20-40-60%-hoz tartozol.
- **Badge-ket** szintén a 4.forduló után kapsz majd először.
- Ha egyszerre több böngészőből, több ablakban vagy több eszközről megnyitod ugyanazt a feladatlapot, **nem tudjuk vállalni** az adatmentéssel kapcsolatban esetlegesen felmerülő anomáliákért a felelősséget!
- A hét forduló során az egyes kategóriákban (de nem feltétlenül mindegyikben) **könnyű-közepes-nehéz kérdésekkel** egyaránt találkozhatsz majd.

Jó versenyzést kívánunk!

Felhasznált idő: 35:00/35:00 **Elért pontszám:** 0/30

1. feladat 0/5 pont

Test drivers

At CMT we generate fake drivers for testing activities. We will use a simplified data model to represent a driver:

```
from pydantic import BaseModel
class Driver(BaseModel):
   id: str
   driver_type: str
   company_id: str
   phone_number: str
```

The ***get_mock_driver(driver_type: int)*** function implements logic to select the appropriate company_id:

- If the driver_type is 42 the company_id should be 'Z'
- Otherwise if the driver_type is even then the company_id should be 'A'
- except if it contains 3, in this case the company_id should be 'D'
- if the driver_type is not even, the company_id should be 'B'
- but in this case if the driver_type contains 4, the company_id will be 'C'
- The type should be between 0 and 99 inclusive, otherwise it throws an Exception with the following message: Not a valid type: {type}.

We implemented the following test cases for this get_mock_driver function. **Which are the ones passing?**

Válaszok

- [x]
```
def test_i(self):
    with self.assertRaises(Exception) as e:
        driver = get_mock_driver(100)
        assert e.exception.args[0] == 'Not a valid type: 100'
```
Ez a válasz helyes, de nem jelölted meg.

- [ ]
```
def test_ii(self):
    driver = get_mock_driver(2)
    assert driver.company_id == 'a'
```

- [ ]
```
def test_iii(self):
    driver = get_mock_driver(103)
    assert driver.company_id == 'C'
```

- [x]
```
def test_iv(self):
    driver = get_mock_driver(42)
    assert driver.company_id == 'Z'
```
Ez a válasz helyes, de nem jelölted meg.

- [ ]
```
@patch('get_driver.get_mock_driver')
def test_v(self, mock):
    get_mock_driver.return_value = 'A'
    driver = get_mock_driver(3)
    assert driver.company_id == 'A'
```

Magyarázat

A valid implementation of the get_mock_driver function:

```
def get_mock_driver(type: int):
    if type < 0 or type > 99:
        raise Exception(f'Not a valid type: {type}')

    company = ''

    # company selector
    if type % 2 == 0:
        company = 'A'
        if '3' in str(type):
```

```
            company = 'D'
    else:
        company = 'B'
        if '4' in str(type):
            company = 'C'

    if type == 42:
        company = 'Z'


    return Driver(id=str(uuid.uuid4()), driver_type=type, company_id=company)
```

FAILED test_driver.py::TestDriver::test_ii - AssertionError: assert 'A' == 'a'

FAILED test_driver.py::TestDriver::test_iii - Exception: Not a valid type: 103

FAILED test_driver.py::TestDriver::test_v - AssertionError: assert 'B' == 'A'

2. feladat 0/10 pont

Registration

At CMT we have a class to represent Users. We will use a simplified data model that contains the email and the password parameters and has two methods:

```
def save_in_db(self) -> 'User'
def send_email(self) -> None
```

A standalone registration method receives the email and password values and creates the user, stores it in the database and sends a registration confirmation email.

```
def register(email: str, password: str) -> 'User'
```

You have the following Test Case template:

```
@patch.object(User, 'send_email')
@patch.object(User, 'save_in_db')
def test_i(self, mock_send_email, mock_save_in_db):

    send_email_mock = MagicMock()
    with patch.object(User, 'send_email', send_email_mock):
        mock_save_in_db.return_value = User('dummy@email.com', 'dummy_password')
        user = register('ithon@email.com', 'password')

    # Place below the asserts
```

Your task is to select the assert or asserts from the list to make sure the test will pass.

Válasz

- [x]
```
assert user.send_email.call_count == 1
```
Ez a válasz helyes, de nem jelölted meg.

```
assert user.email == 'dummy@email.com'
```

```
assert user.email == 'ithon@email.com'
```

```
assert user.password == 'password'
```

```
assert user.password == 'dummy_password'
```

```
assert user.save_in_db.call_count == 1
```

Magyarázat

There are two tricks in this example:

The save_in_db function will return with self (based on the return type)

The test case function is tricked: def test_i(self, mock_send_email, mock_save_in_db) because the mock objects are replaced. The valid implementation should look like this: def test_vii(self, mock_save_in_db, mock_send_email):

If the mocks are not replaced, the valid test would look like this:

```
    @patch.object(User, 'send_email')
    @patch.object(User, 'save_in_db')
    def test_i(self, mock_save_in_db, mock_send_email):

        send_email_mock = MagicMock()
        with patch.object(User, 'send_email', send_email_mock):
            mock_save_in_db.return_value = User('dummy@email.com', 'dummy_password')

            user = register('ithon@email.com', 'password')

        assert user.password == 'dummy_password'
        assert user.email == 'dummy@email.com'
        assert user.save_in_db.call_count == 1
```

Valid implementation of User:

```
class User:
    def __init__(self, email, password):
        self.email = email
        self.password = password

    def save_in_db(self) -> 'User':
        print(f'Save user in the database... {self.email}')

        return self

    def send_email(self) -> None:
        print('Send email')

Valid implementation of register:
from user import User
```

```
def register(email: str, password: str) -> User:
    # do some magic with email and password

    user = User(email, password).save_in_db()

    user.send_email()

    return user
```

3. feladat 0/15 pont

Chemical descriptors

Given the class ChemicalCompound representing 2D chemical compounds and function calculate_pKa(molecule: ChemicalCompound), where calculate_pKa returns a floating point number descriptor while doing graph traversals and matrix calculations on the input compound without using any other resources.

Which of the following options should be used to print all calculated pKa values with a molecule identifier on the standard input in order to boost time performance of the calculation?

Válasz

```
from typing import List
...

if __name__ == "__main__":
    molecules: List[ChemicalCompound] = [...]
    calculated_pKa = {molecule.id: calculate_pKa(molecule) for molecule in molecules}
    print(calculated_pKa)
```

```
import multiprocessing
...

def calculate_and_print(molecule: ChemicalCompound):
    calculated_pKa = calculate_pKa(molecule)
    print(f"{molecule.id}, {calculated_pKa}")

if __name__ == "__main__":
    molecules: List[ChemicalCompound]  = [...]
    with multiprocessing.Pool() as pool:
        pool.map(calculate_and_print, molecules)
```

Ez a válasz helyes, de nem jelölted meg.

```
import asyncio
...

async def calculate_and_print(molecule: ChemicalCompound):
    calculated_pKa = calculate_pKa(molecule)
    print(f"{molecule.id}, {calculated_pKa}")

async def calculate_all_pKa(molecules: List[ChemicalCompound]):
    tasks = []
    for molecule in molecules:
        task = asyncio.ensure_future(calculate_and_print(molecule))
        tasks.append(task)
```

```
    await asyncio.gather(*tasks, return_exceptions=True)

if __name__ == "__main__":
    molecules: List[ChemicalCompound] = [...]
    asyncio.get_event_loop().run_until_complete(calculate_all_pKa(molecules))
```

```
import gevent
...

def calculate_and_print(molecule: ChemicalCompound):
    calculated_pKa = calculate_pKa(molecule)
    print(f"{molecule.id}, {calculated_pKa}")

if __name__ == "__main__":
    molecules: List[ChemicalCompound] = [...]
    threads = []
    for molecule in molecules:
        threads.append(gevent.spawn(calculate_and_print, molecule))
    gevent.joinall(threads)
```

Magyarázat

Based on the description the calculate_pKa function is CPU heavy and does not use IO. This means we need to optimize for CPU.

List executes the calculations in sequence, so the total execution time is the sum of each calculation plus the execution of the cycle.

asyncio makes execution of each calculation async. Since the calculation does not use IO and async methods are still executed on the same process and CPU core we do not gain any execution time with this solution, it might be even slower than answer A based on the version of asyncio.

gevent uses greenlets and is essentially sequential according to its documentation.

multiprocessing makes use of multiple CPU cores (and in most computers we have multiple cores nowadays). With this solution we can reduce execution time to a fragment of the sequential solutions.