PYTHON IN CLOUD (ANGOL NYELVŰ)

4. forduló

CAMBRIDGE MOBILE TELEMATICS

A kategória támogatója: Cambridge Mobile Telematics (CMT)

Ismertető a feladathoz

A 4. forduló után elérhetőek lesznek a helyezések %-os formában: azaz kiderül, hogy a kategóriában a versenyzők TOP 20% - 40% -60% -ához tartozol-e!

Szeretnénk rá felhívni figyelmedet, hogy a játék nem Forma-1-es verseny! Ha a gyorsaságod miatt kilöködöd a rendesen haladó versenyzőket, kizárást vonhat maga után!

Felhasznált idő: 00:00/40:00

Elért pontszám: 0/15

1. feladat 0/5 pont

OOP

Which OO principle is violated by the following code?

```
class MySuperBase:
    _counter: int

    def __init__(self):
        self._counter = 0

    def update(self):
        self._counter += 1

    def __str__(self):
        return f'counter:{self._counter}'

class MyClass(MySuperBase):
    def update(self, num: int):
        self._counter += num
```

Válasz

[✓ Liskov substitution principle (LSP)]
Ez a válasz helyes, de nem jelölted meg.

Open-closed principle (OCP)

No principle is violated

○ both LSP and OCP

Magyarázat

The Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application. Because the MyClass's update method has different parameter signature then the original class's update, you can not use the MyClass as a direct replacement of the MySuperBase class.

2. feladat 0/10 pont

Iterate 'em all!

Which of the following generators will return an ascending list of unique numbers for an iterator of random numbers as input?

Válasz

○ A.

```
def unique(iter: Iterator):
    s = set()
    for item in iter:
        if item in s:
            continue
        else:
            s.add(item)
            yield item
```

○ B.

```
def unique(iter: Iterator):
    last_one = -1

    for item in iter:
        if item < last_one:
            continue
        else:
            last_one = item
            yield item
```

● C.

```
def unique(iter: Iterator):
    last_one = None

    for item in iter:
        if last_one is not None and item <= last_one:
            continue
        else:
            last_one = item
            yield item
```

Ez a válasz helyes, de nem jelölted meg.

○ D.

```
def unique(iter: Iterator):
   s = set()

   for item in iter:
       if s and item <= s.pop():
           continue
       else:
           s.add(item)
           yield item
```

Magyarázat

Since we need an ascending list of unique items, we have to make sure that each subsequent item is larger than the previous one.

Answer A only fulfills the unique requirement, does not guarantee an ascending list.

Answer B does not guarantee uniqueness and only works for numbers that are larger than or equals -1.

Answer D would only work, if the set.pop() command would result in the last item of the set, but it returns a pseudo-random element making the result non-deterministic.

3. feladat 0/0 pont

Topological sort

Your goal is to implement the blockers_and_blocked function below: for a given task and a set of dependencies, return all tasks that must be completed before the task, and all tasks that are blocked by the task.

Example runs:

```
blockers_and_blocked("a", [("a", "b"), ("b", "c")])
(set(), {"b", "c"})
blockers_and_blocked("b", [("a", "b"), ("b", "c")])
({"a"}, {"c"})
blockers_and_blocked("c", [("a", "b"), ("b", "c")])
({"a", "b"}, set())
blockers_and_blocked("c", [("a", "b"), ("b", "c"), ("e", "f")])
({"a", "b"}, set())
blockers_and_blocked("f", [("a", "b"), ("b", "c")])
(set(), set())
```

Code skeleton:

```
import random
from typing import Callable, List, Set, Tuple

Dependency = Tuple[str, str]

def blockers_and_blocked(
    task: str, dependencies: List[Dependency]
) -> Tuple[Set[str], Set[str]]:
    """Parameters
    ----------
    task
        The test for which we compute blocking and blocked tasks.
    dependencies
        A list of pairs of tasks, where the first element blocks the
        second."""
    raise NotImplementedError()

def generate_cases(seed: int):
    """Generate tests cases using the given seed."""
    rand = random.Random(seed)

    for _ in range(100):
        num_tasks = rand.randint(0, 100)
        tasks = list(map(str, range(num_tasks)))
        dependencies = []

        rand.shuffle(tasks)

        dependencies = [
            (s, t)
            for i, s in enumerate(tasks)
            for t in tasks[i + 1 :]
            if rand.random() > 0.5
        ]

        yield rand.choice(tasks), dependencies

def checksum(
    implementation: Callable[[str, List[Dependency]], Tuple[Set[str], Set[str]]],
    seed: int,
) -> int:
    """Compute a checksum of an implementation of `blockers_and_blocked` and a random seed."""

    result = 0
    for task, dependencies in generate_cases(seed):
        blockers, blocked = implementation(task, dependencies)

        result = hash((result, tuple(sorted(blockers)), tuple(sorted(blocked))))

    return result


if __name__ == "__main__":
    #
    # To help testing, checksum(blockers_and_blocked, 1) should return 2150572700622488023
    #
    seed_1_answer = 2150572700622488023
```

As a proof of your implementation, please provide what does this call return?

checksum(0, blockers_and_blocked)

Válasz

A helyes válasz:

-3560732291628200414

Magyarázat

Kedves Versenyzők!

A feladatot 0 pontosra állítottuk, mivel abban sajnos több hiba volt:

- *a megadott checksum függvény implementációja hibás*
- *a generate_cases hibás*
- *fordítva került kiírásra két paraméter*

Elnézést kérünk a kellemetlenségért!

(2022.11.09.)

To find tasks blocked by a task, we can construct the dependency graph and perform a depth-first search from the starting task. Any tasks found in the search are blocked by this task. To find blockers, we can repeat this process on the reversed dependency graph.