PYTHON IN CLOUD (ANGOL NYELVŰ)

3. forduló

CAMBRIDGE MOBILE TELEMATICS

A kategória támogatója: Cambridge Mobile Telematics (CMT)

Ismertető a feladathoz

A 3.forduló feladatait a hosszú hétvége miatt kivételesen szerda (11.02.) éjfélig tudod megoldani!

Érdemes ebben a fordulóban is játszanod, mert a következő forduló kezdetekor, 11.03-án 18 órától kiosztjuk az 1.-2.-3. fordulóban megszerzett badgeket!

A verseny közben az alábbi teljesítményeket díjazzuk:

- fordulógyőztes
- átlagnál jobb időeredmény
- átlag feletti pontszám
- hibátlan forduló

Szeretnénk rá felhívni figyelmedet, hogy az egyszer megkapott badge-eket nem vonjuk vissza, akkor sem, ha esetleg az adott fordulóban a visszajelzések alapján változások vannak.

Jó játékot!

Felhasznált idő: 13:47/40:00 **Elért pontszám:** 0/25

1. feladat 0/5 pont

Generic functions

How do you create generic functions in python?

Válasz

```
@singledispatch
def fun(arg, verbose=False):
    if verbose:
        print("my print: ", end=" ")
        print(arg)

@fun.register
def _(arg: int, verbose=False):
    if verbose:
```

```
        print("int print:", end=" ")
    print(arg)

@fun.register
def _(arg: list, verbose=False):
    if verbose:
        print("List print:")
    for i, elem in enumerate(arg):
        print(i, elem)
```

Ez a válasz helyes, de nem jelölted meg.

```
def fun(arg: int, verbose=False):
    if verbose:
        print("Let me just say,", end=" ")
        print(arg)

def fun(arg: list, verbose=False):
    if verbose:
        print("Enumerate this:")
    for i, elem in enumerate(arg):
        print(i, elem)
```

```
def fun_int(arg=int, verbose=False):
    if verbose:
        print("Let me just say,", end=" ")
        print(arg)

def fun_list(arg=list, verbose=False):
    if verbose:
        print("Enumerate this:")
    for i, elem in enumerate(arg):
        print(i, elem)
```

Ez a válasz helytelen, de megjelölted.

```
def fun(arg=int, verbose=False):
    if verbose:
        print("Let me just say,", end=" ")
        print(arg)

def fun(arg=list, verbose=False):
    if verbose:
        print("Enumerate this:")
    for i, elem in enumerate(arg):
        print(i, elem)
```

Magyarázat

The @singledispatch decorator is a form of generic function dispatch where the implementation is chosen based on the type of a single argument. The rest of the codes are syntactically correct, however none of them gives generic function. Definition of generic function: A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

2. feladat 0/5 pont

Method Resolution order

What is the MRO (Method Resolution order) for the MyAwesomeClass?

```
class MySuperClass1:
    def text(self):
        return 'Hey'

class MySuperClass2(MySuperClass1):
    def text(self):
        return 'Ho'

class MySuperClass3:
    def text(self):
        return 'Hi'

class MyAwesomeClass(MySuperClass3, MySuperClass2):
    def text(self):
        return super().text()
```

Válasz

- (<class '__main__.MyAwesomeClass'>,<class '__main__.MySuperClass3'>, <class '__main__.MySuperClass2'>, <class '__main__.MySuperClass1'>, <class 'object'>)
  Ez a válasz helyes, de nem jelölted meg.
- (<class '__main__.MyAwesomeClass'>,<class '__main__.MySuperClass2'>, <class '__main__.MySuperClass3'>, <class '__main__.MySuperClass1'>, <class 'object'>)
  Ez a válasz helytelen, de megjelölted.
- (<class '__main__.MyAwesomeClass'>,<class '__main__.MySuperClass1'>, <class '__main__.MySuperClass2'>, <class '__main__.MySuperClass3'>, <class 'object'>)
- (<class '__main__.MySuperClass3'>,<class '__main__.MySuperClass2'>, <class '__main__.MySuperClass1'>, <class '__main__.MyAwesomeClass'>, <class 'object'>)

Magyarázat

In Python, the MRO is from bottom to top and left to right. This means that, first, the method is searched in the class of the object. If it's not found, it is searched in the immediate super class. In the case of multiple super classes, it is searched left to right, in the order by which was declared by the developer.

3. feladat 0/15 pont

B-Trees (https://en.wikipedia.org/wiki/B-tree) are incredibly handy data structures for helping to keep large amounts of data organized and quickly accessible. They share some similarities with Binary Search Trees, but are a bit more complicated to be able to hold more keys in each node and to have self-balancing behavior.

For this problem, we've provided a very basic implementation of a B-Tree which can have keys inserted into it. However, we've notably left certain other features missing, most notably any sort of retrieval (deletion won't be relevant to this problem).

To complete this challenge, you'll need to be able to, given a tree and a particular key, retrieve the node (with its children, etc. intact) within the tree that contains that node. This should be implemented in the function get_node_with_key, but you are free to add to the B-Tree classes or any helper functions you may need.

Provided your implementation is in the above function, running the provided checksum function with your implementation and the correct seed (0) will give you the answer you'll need to input here (this call is provided in the code). For testing purposes, we provide the answer for the seed 1 case.

```
class BTreeNode:
    def __init__(self, parent, num_children):
        self.keys = []          # values contained in this node
        self.n = num_children
        self.children = []
        self.parent = parent

    def is_leaf(self):
        return not self.children

    def is_root(self):
        return self.parent is None

    def overflow(self, left, right, key):
        # easy, we just need to add the new key and readjust the children
        old_child_idx = self.children.index(left)
        self.keys.insert(old_child_idx, key)
        self.children.insert(old_child_idx + 1, right)
        if len(self.keys) >= self.n:
            # we're going to overflow higher
            median_idx = len(self.keys) // 2
            median_key = self.keys[median_idx]

            # we will retain the left half in this node
            # and give the other half to the other node
            if self.is_root():
                # BTrees are funky, in that they grow at the root
                self.parent = BTreeNode(None, self.n)
                self.parent.children.append(self)
            sibling = BTreeNode(self.parent, self.n)
            sibling.keys = self.keys[median_idx+1:]
            sibling.children = self.children[median_idx+1:]
            for c in sibling.children:
                c.parent = sibling
            self.keys = self.keys[:median_idx]
            self.children = self.children[:median_idx+1]
            self.parent.overflow(self, sibling, median_key)

    def insert(self, key):
        if key in self.keys:
            # already in this node
            return

        if self.is_leaf():
            if len(self.keys) < (self.n - 1):
                # we have room
                for i, k in enumerate(self.keys):
                    # there are no children to worry about
                    if k > key:
                        self.keys.insert(i, key)
                        break
                else:
                    self.keys.append(key)

            else:
                all_keys = sorted(self.keys + [key])
                median_idx = len(all_keys) // 2
                median_key = all_keys[median_idx]
```

```
                if self.is_root():
                    self.parent = BTreeNode(None, self.n)
                    self.parent.children.append(self)

                # we will retain the left half in this node
                # and give the other half to the other node
                sibling = BTreeNode(self.parent, self.n)
                sibling.keys = all_keys[median_idx+1:]
                self.keys = all_keys[:median_idx]

                self.parent.overflow(self, sibling, median_key)
        else:
            for i, k in enumerate(self.keys + [float('inf')]):
                if k > key:
                    # this is the correct spot for the child
                    self.children[i].insert(key)
                    return

    def checksum(self):
        # recursive checksum
        node_hash = hash(tuple(self.keys))
        if self.children:
            children_hash = hash(tuple(child.checksum() for child in self.children))
            node_hash = hash((node_hash, children_hash))
        return node_hash

    def print(self, l=0):
        print('\t'*l, f"level {l} {self.keys}")
        for c in self.children:
            c.print(l+1)

class BTree:
    def __init__(self, order):
        self.order = order
        self.root = None

    def insert(self, key):
        if not self.root:
            self.root = BTreeNode(None, self.order)
        self.root.insert(key)
        if not self.root.is_root():
            # we grew
            self.root = self.root.parent

    def print(self):
        if not self.root:
            print("None")
        else:
            self.root.print(0)


def find_node_with_key(tree: BTree, key: int) -> BTreeNode:
    """
    For a given B-Tree, which is a special type of search tree, return
    the node containing the provided key.  You can assume in all cases that
    the key exists in one of the nodes of the tree passed in
    Parameters
    __________
    tree
        A BTree object containing at least one key
    key
        A value to look for within the tree
    """
```

```
    raise NotImplementedError()


def generate_cases(seed):
    # DO NOT MODIFY
    rand = random.Random(seed)

    for i in range(100):
        tree = BTree(1 + 2*rand.randint(1, 4))
        tree_keys = list(range(1000))
        rand.shuffle(tree_keys)
        tree_keys = tree_keys[:rand.randint(20, 100)]
        for key in tree_keys:
            tree.insert(key)
        yield tree, rand.choice(tree_keys)


def checksum(
    implementation,
    seed
):
    # DO NOT MODIFY
    result = 0
    for tree, key in generate_cases(seed):
        node = implementation(tree, key)
        result = hash((result, node.checksum()))

    return result

if __name__ == "__main__":
    #
    # To help testing, checksum(find_node_with_key, 1) should return -2803493182871350012
    #
    seed_1_answer = -2803493182871350012

    #
    # Your final answer is the result of the next line
    #
    print(checksum(find_node_with_key, 0))
```

Válaszok

A helyes válasz:

8427739854948104963

-1095545274067579368

Magyarázat

A helyes megoldás függ a használt verziószámtól: python 3.8-nál és felette 8427739854948104963 a helyes megoldás, python 3.7-nél -1095545274067579368.

In a B-Tree, in a given node the keys are stored in sorted order, and the values in each subtree under that node are bounded by the values of the keys in that node. E.g. if there is a node containing the keys 3, 56, 102, then (respecting the similar constraints from the parents) the leftmost child will contain anything less than 3, the next one will contain things between 3 and 56 (exclusive), then the next between 56 and 102 (exclusive), and finally anything bigger than 102 if the node is full. For merely traversing the tree, dealing with the self-balancing rules is irrelevant.

Traversing the B-tree is then a lot like traversing a binary tree in concept. One finds the leftmost key in the current node that is greater than the desired key, then that corresponds to the appropriate child to dive into next. Once you find a node containing the key, if there is one, return the node itself. Here is a sample implementation (implemented as a method of the `BTreeNode` class):

```
    def find_node(self, key):
        for i, k in enumerate(self.keys + [float('inf')]):
            if key == k:
                return self
            if key < k:
                return self.children[i].find_node(key)
```