PYTHON IN CLOUD (ANGOL NYELVŰ)

5. forduló

CAMBRIDGE MOBILE TELEMATICS

A kategória támogatója: Cambridge Mobile Telematics (CMT)

Ismertető a feladathoz

Felhasznált idő: 25:00/25:00 **Elért pontszám:** 5/20

1. feladat 0/5 pont

Is it pharmaceutical?

Assume we have a method that can evaluate whether a chemical compound is fit to be a pharmaceutical: is_pharma(compound: Compound).

Given a list of compounds, which option removes non-pharmaceutical from the compounds list?

Válasz

A.

```
for x in compounds:
   if not is_pharma(x):
       compounds.remove(x)
```

B.

```
for x in compounds:
   if not is_pharma(x):
       del x
```

Ez a válasz helytelen, de megjelölted.

C.

```
for x in compounds if is_pharma(x):
   compounds.remove(x)
```

D.

```
compounds = [x for x in compounds if is_pharma(x)]
```

Ez a válasz helyes, de nem jelölted meg.

E.

```
new_compounds = []
for x in compounds:
   if is_pharma(x):
       new_compounds.append(x)
```

Magyarázat

The task is about removing items from a list while iterating on it.

Answer A is incorrect as the result may fail with what may be unexpected results depending on what type of the collection compounds is. CPython list is implemented with dynamic arrays which do not support such removals.

Answer B does not delete items from the list.

Answer C is not a valid python syntax.

Answer E collects the correct answers in the list new_compounds, but it is not used as the result is expected to be in the compounds list.

2. feladat 0/10 pont

Log exceptions

Assume we are creating a public python library for reading and writing in parquet data format. We have a method for file access that can raise 2 types of exception: FileNotFoundError and PermissionError.

We would like to handle both of these the same way in our logs for the user of our library. Which of the following exception handlings is the preferred way?

Válasz

A.

```
def read_file(path: str):
   try:
       ...
   except:
       logger.error("File at [" + path + "] could not be accessed!")
```

B.

```
def read_file(path: str):
   try:
       ...
   except FileNotFoundError, PermissionError, e:
       logger.error(f"File at [{path}] could not be accessed!")
```

Ez a válasz helytelen, de megjelölted.

C.

```
def read_file(path: str):
    try:
        ...
    except (FileNotFoundError, PermissionError) as e:
        logger.error("File at [%s] could not be accessed!", path)
        raise
```

Ez a válasz helyes, de nem jelölted meg.

D.

```
def read_file(path: str):
    try:
        ...
    except (FileNotFoundError, PermissionError) as e:
        logger.error("File at [%s] could not be accessed!", path)
```

E.

```
def read_file(path: str):
    try:
        ...
    except Exception:
        logger.error(f"File at [{path}] could not be accessed!")
        raise
```

Magyarázat

This exercise is about creating a public library.

Since best practice is catching the most specific type of exception we should not use except: or except Exception:, meaning answers A and E incorrect.

Answer B is only accepted by older interpreters, even then it does not catch PermissionError, only FileNotFoundError.

Answer D silently consumes an exception, that is the only source of help for the user of the library on file access issues, hence is a wrong answer.

3. feladat 5/5 pont

Company builder

At CMT we have a data model to represent a Company. We are looking to implement a Builder Pattern to easily create new companies. Your task is to select the valid code snippets to properly finish the implementation.

```
from abc import ABC, abstractmethod

class Company():
    def __init__(self) -> None:
        self.__company_id = ''
        self.__title = ''

    def __set_company_id(self, company_id: str) -> None:
        self.__company_id = company_id

    def __get_company_id(self) -> str:
```

```
        return self.__company_id

    def __set_title(self, title: str) -> None:
        self.__title = title

    def __get_title(self) -> str:
        return self.__title

    title = property(__get_title, __set_title)
    company_id = property(__get_company_id, __set_company_id)

    def properties(self) -> None:
        print(f'{self.__title} company with {self.__company_id} company_id')

class ICompanyBuilder(ABC):
    # missing implementation

class ACompanyBuilder(ICompanyBuilder):
    # missing implementation

class BCompanyBuilder(ICompanyBuilder):
    # missing implementation

class CompanyFactory():
    # missing implementation
```

Make sure that the following Test Case will pass:

```
class TestCompany(TestCase):

    def test_company(self):
        factory = CompanyFactory()

        factory.set_builder(ACompanyBuilder())
        company = factory.get_company()
        assert company.company_id == '1'
        assert company.title == 'A'

        factory.set_builder(BCompanyBuilder())
        company = factory.get_company()
        assert company.company_id == '2'
        assert company.title == 'B'
```

Válasz

i.)

```
class ICompanyBuilder(ABC):
    @abstractmethod
    def build(self):
        "build"
class ACompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company()
        company.title = 'A'
        company.company_id = '2'
        return company
class BCompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company()
        company.title = 'B'
```

```
        company.company_id = '1'
        return company
class CompanyFactory():
    __builder: ICompanyBuilder = None
    def set_builder(self, builder: ICompanyBuilder) -> None:
        self.__builder = builder
    def get_company(self) -> Company:
        company = self.__builder.build()
        return company
```

ii.)

```
class ICompanyBuilder(ABC):
    @abstractmethod
    def build(self):
        "build"
class ACompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company()
        company.title = 'A'
        company.company_id = '1'
        return company
class BCompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company()
        company.title = 'B'
        company.company_id = '2'
        return company
class CompanyFactory():
    __builder: ICompanyBuilder = None
    def set_builder(self, builder: ICompanyBuilder) -> None:
        self.__builder = builder
    def get_company(self) -> Company:
        company = self.__builder.build()
        return company
```

Ez a válasz helyes, és meg is jelölted.

iii.)

```
class ICompanyBuilder(ABC):
    @abstractmethod
    def build(self):
        "build"
class ACompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company()
        company.title = 'A'
        company.company_id = '1'
        return company
class BCompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company()
        company.title = 'B'
        company.company_id = '2'
        return company
class CompanyFactory():
    __builder: ICompanyBuilder = ICompanyBuilder()
    def set_builder(self, builder: ICompanyBuilder) -> None:
        self.__builder = builder
    def get_company(self) -> Company:
        company = self.__builder.build()
```

```
        return company
```

iv.)

```
class ICompanyBuilder(ABC):
    @abstractmethod
    def build(self):
        "build"
class ACompanyBuilder(ICompanyBuilder):
    __company = None
    def build(self) -> Company:
        self.__company = Company()
        self.__company.title = 'A'
        self.__company.company_id = '1'
class BCompanyBuilder(ICompanyBuilder):
    __company = None
    def build(self) -> Company:
        self.__company = Company()
        self.__company.title = 'B'
        self.__company.company_id = '2'
class CompanyFactory():
    __builder: ICompanyBuilder = None
    def set_builder(self, builder: ICompanyBuilder) -> None:
        self.__builder = builder
    def get_company(self) -> Company:
        self.__builder.build()
        company = self.__builder.__company
        return company
```

v.)

```
class ICompanyBuilder(ABC):
    @abstractmethod
    def build(self):
        return None
class ACompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company
        company.title = 'A'
        company.company_id = '1'
        return company
class BCompanyBuilder(ICompanyBuilder):
    def build(self) -> Company:
        company = Company
        company.title = 'B'
        company.company_id = '2'
        return company
class CompanyFactory():
    __builder: ICompanyBuilder = None
    def set_builder(self, builder: ICompanyBuilder) -> None:
        self.__builder = builder
    def get_company(self) -> Company:
        company = self.__builder.build
        return company
```

Magyarázat

i.) not valid, because the company_ids are replace in the A and B company builder

ii.) valid, and pass the test case

iii.) not valid, because TypeError: Can't instantiate abstract class ICompanyBuilder with abstract methods build

iv.) __company is not accessible, do not pass the Test Case, do not pass the the return restriction, will return None instead of Company, also not a “nice” implementation of builder pattern

v.) AttributeError: 'function' object has no attribute 'company_id'