Python (BSU FAMCS Fall'19)

Seminar 4

Advisor: Dzmitryi Kasitsyn

Task 1. (1 point). Implement a decorator `handle_error` to handle exceptions in a function regarding decorator parameters. Parameters are the next:

- `re_raise` – a flag to control whether an exception will be re-raised from a function, `True` by default. All exceptions that aren't inherited from `exc_type` should be re-raised unconditionally;
- `log_traceback` – exception traceback will be logged if the flag is set to `True` (by default). All exceptions that aren't inherited from `exc_type` should not be processed;
- `exc_type` – exception base type or a non-empty tuple of exception base types that are handled by the decorator (`Exception` by default);
- `tries` – number of times function has to be invoked again with the same parameters if it raises an exception (default value `1` means no repeats). Check the value of tries provided as infinite tries are not permitted (e.g. `None` or negative integer values);
- `delay` – a delay between tries in seconds (may be `float`, by default it's `0`);
- `backoff` – a value that a `delay` is multiplied by from attempt to attempt (by default `1`, see an example below).

Note the usage of a module global `logger` object is an ordinary practice. The logger is an instance of `logging.Logger` basically.

Save the decorator in *error_handling.py* file.

Example 1

```
# suppress exception, log traceback
@handle_error(re_raise=False)
def some_function():
    x = 1 / 0  # ZeroDivisionError

some_function()
print(1)  # line will be executed as exception is suppressed
```

Example 2

```
# re-raise exception and doesn't log traceback as exc_type doesn't match
@handle_error(re_raise=False, exc_type=KeyError)
def some_function():
    x = 1 / 0  # ZeroDivisionError

some_function()
print(1)  # line won't be executed as exception is re-raised
```

Example 3

Let suppose that `random.random()` function consequently produces `0.2`, `0.5`, `0.3` values. Thus the decorator invokes an original `some_function`, handles it, waits for `0.5` seconds, tries again, waits for `1` more second, tries again and finally re-raises an exception.

```
import random

@handle_error(re_raise=True, tries=3, delay=0.5, backoff=2)
def some_function():
    if random.random() < 0.75:
        x = 1 / 0  # ZeroDivisionError

some_function()
```

Task 2. (0.5 points). Implement a context manager `handle_error_context` that is idiomatically similar to `handle_error` decorator from the task above and intended to handle exceptions depending on the next parameters:

- `re_raise` – a flag to control whether an exception will be re-raised from a function, `True` by default. All exceptions that aren't inherited from `exc_type` should be re-raised unconditionally;
- `log_traceback` – exception traceback will be logged if the flag is set to `True` (by default). All exceptions that aren't inherited from `exc_type` should not be processed;
- `exc_type` – exception base type or a non-empty tuple of exception base types that are handled by the decorator (`Exception` by default);

The code of `handle_error` decorator is supposed to be re-used to avoid code duplication implementing your context manager. Also try to avoid class based context manager implementation.

Save your solution in *error_handling.py* file.

Example

```
# log traceback, re-raise exception
with handle_error_context(log_traceback=True, exc_type=ValueError):
    raise ValueError()
```

Task 3. (1 point). Implement a metaclass `BoundedMeta` that limits the number of class instances created.

Provide a parameter `max_instance_count` to set maximal number of instances value (1 by default). Raise an exception `TypeError` trying to create a new instance over the limit. Number of instances will be supposed to be unlimited if `max_instance_count` value equals `None`.

In an example below class `C` has `BoundedMeta` metaclass so no more than 2 instances may be created.

Save your metaclass implementation in *functional.py* file.

BoundedMeta class boilerplate

```
class C(metaclass=BoundedMeta, max_instance_count=2):
    pass

c1 = C()
c2 = C()

try:
    c3 = C()
except TypeError:
    print('everything works fine!')
else:
    print('something goes wrong!')
```

Task 4. (1 point). Implement a class `BoundedBase` that has an abstract class method providing a value of maximal number of class instances that are permitted to create.

Name the method `get_max_instance_count`.

As in the previous task raise an exception `TypeError` trying to create a new instance over the limit. Number of instances will be supposed to be unlimited if `get_max_instance_count` returns `None`.

In an example below only one instance of class `D` inherited from `BoundedBase` class is allowed to be created.

Save your class implementation in *functional.py* file.

BoundedBase class boilerplate

```
class D(BoundedBase):
    @classmethod
    def get_max_instance_count(cls):
        return 1
```

```
d1 = D()

try:
    d2 = D()
except TypeError:
    print('everything works fine!')
else:
    print('something goes wrong!')
```

Task 5. (0.5 points). Implement a function that returns the number of times it has been called.

Global variables are not permitted. In other words all code related to the task may contain only a function definition leading with a keyword `def`.

Name your solution function `smart_function` and save it into *functional.py* file.

Example

```
for real_call_count in range(1, 5):
  assert f() == real_call_count
```