1 With Statement

Generally, `with` statement is used to wrap the execution of a block with With Statement Context Managers. It simplifies the management of common resources. `with` ensures proper acquisition and release of resources, so helps avoiding bugs and leaks. Previously, the similar usage might be achieved through `try...finally...` block, `with` make it more simple and readable.

[ ]:
```
# basic structure
with expression as [variable]:  # result in an object that supports the context␣
 ↪manager protocol
    # such that, has _enter_() / _exit_() methods
    # The object's _enter_ is called before block execution
    # also could return a value bound to [variable]
    block_execution
    # after finish block_execution, _exit_() is called

# file handling
with open('path', 'w') as file:
    file.write('hello world!')  # do not need to manually close file

# in user defined objects
class OwnWriter(object):
    def _init_(self, file_name):
        self.file_name = file_name

    def _enter_(self):
        self.file = open(self.file_name, 'w')
        return self.file

    def __exit__(self):
        self.file.close()

with OwnWriter("file_name") as file:
    file.write("hello world")
```

2 Decorator

Functions are the first class objects. Decorators make it possible to wrap a function to add more behaviors, but do not need to permanently modifying it.

[10]:
```
def this_decorator(func):  # define a decorator
    print("before execution")
    def inner(*args, **kw):
        value = func(*args, **kw)
        print("after execution")
```

```
        return value
    return inner
```

[11]:
```
@this_decorator
def now():  # equal to now = this_decorator(now); now points to a new funciton
 ↪inner
    print("2020") # before inner, before execution is printed
```

```
before execution
```

[12]:
```
now()
```

```
2020
after execution
```

[20]:
```
@this_decorator
def sum_number(a, b):
    print("execution")
    return a + b
```

```
before execution
```

[23]:
```
print("sum=", sum_number(5,6))  # value is returned after execution
```

```
execution
after execution
sum= 11
```

3 Iterators

an object that is used to iterate over iterable objects.

[25]:
```
iterable_list = [1, 2, 3, 4, 5]
iterable_obj = iter(iterable_list)  # iter(iterable) is called to initialze an
 ↪iterator, returns an iteratot object
while True:
    try:
        item = next(iterable_obj)  # next(iterator) returns the next value. When
 ↪end, raise StopIteration
        print(item)
    except StopIteration:
        break
```

```
1
2
3
```

```
4
5
```

[26]:
```python
# some built-in iterator

print("list")
l = ["one","two","three"]
for i in l:
    print(i)

print("tuple")
t = ("one","two","three")
for i in t:
    print(i)

print("string")
s = "one"
for i in s:
    print(i)
```

```
list
one
two
three
tuple
one
two
three
string
o
n
e
```

[28]:
```python
# to test whether an object is Iterable, so that we could use iterator
from collections.abc import Iterable
print(isinstance('123', Iterable))
print(isinstance(123, Iterable))
```

```
True
False
```

4 Generator

- `yield` keyword: suspends the function execution and sends value back to the caller, but it will remain the state so that the function could start from where left.
- **Generator Function**: defines as a normal function, but use `yield` instead of `return` when

need to generate the value. **Generator Object**: generator function will return a generator object. Generator object could be iterated by `next` or using iterators.

[29]:
```
def fib(max):  # with `yield`, becomes a generator function
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

[47]:
```
fib(6)  # call the generator funciton, a generator object return
```

[47]:
```
<generator object fib at 0x7f81d18aaa50>
```

[46]:
```
f = fib(6)
while True:
    try:
        print(next(f))  # generator execute when call next, return when yield.␣
 ↪next exectuion will begin from yield
    except StopIteration as e:
        print(e.value)
        break
```

```
1
1
2
3
5
8
done
```

[48]:
```
for i in fib(6):  # using loop to iterate over the generator object
    print(i)
```

```
1
1
2
3
5
8
```

[ ]:

[ ]: