PYTHON TEST - 3.7 (COMPOSITION)

Total points 50/50

Composition in Python

STUDENT NAME *

VIVA

✓ **1.** In Python, *composition* usually refers to: * 1/1

- (●) a) Combining functions to create new behavior ✓
- ( ) b) Combining classes using inheritance
- ( ) c) Overloading operators
- ( ) d) Importing modules

✓ **2.** Function composition in Python is mainly used to: * 1/1

- ( ) a) Increase execution time
- ( ) b) Decrease code readability
- (●) c) Combine smaller functions into a larger one ✓
- ( ) d) Avoid using functions

✓ **3.** Which operator is often associated with mathematical composition of functions? *1/1

- a) +
- b) *
- c) ○ (circle dot) ✓
- d) ->

✓ **4.** In Python, function composition is typically achieved using: * 1/1

- a) Nested function calls ✓
- b) Inheritance
- c) Global variables
- d) File handling

✓ **5.** Example of function composition: * 1/1

```
def square(x): return x*x

def double(x): return 2*x

print(square(double(3)))
```

What is the output?

- a) 36 ✓
- b) 18
- c) 12
- d) 9

✓ **6.** In the above program, square(double(3)) means: * 1/1

- a) Apply square then double
- b) Apply double then square ✓
- c) Both at the same time
- d) Invalid operation

✓ **7.** Composition in Python allows functions to be: * 1/1

- a) Isolated
- b) Combined ✓
- c) Deleted
- d) Rewritten only

✓ **8.** Which Python concept enables composition naturally? * 1/1

- a) Recursion
- b) First-class functions ✓
- c) Encapsulation
- d) Abstraction

✓ **9.** Python treats functions as: * 1/1

- (●) a) Objects ✓
- ( ) b) Modules
- ( ) c) Classes
- ( ) d) Operators

✓ **10.** If functions are objects, they can be: * 1/1

- ( ) a) Passed as arguments
- ( ) b) Returned from other functions
- ( ) c) Assigned to variables
- (●) d) All of the above ✓

✓ **11.** Example: * 1/1

```
def f(x): return x + 2

def g(x): return x * 3

print(f(g(4)))
```

Output?

- (●) a) 14 ✓
- ( ) b) 18
- ( ) c) 12
- ( ) d) 20

✓ **12.** The order of composition f(g(x)) means: * 1/1

- a) Apply f first, then g
- b) Apply g first, then f ✓
- c) Apply both in parallel
- d) None

✓ **13.** Which Python module has tools for functional composition? * 1/1

- a) functools ✓
- b) itertools
- c) math
- d) sys

✓ **14.** Example using lambda: * 1/1

```
compose = lambda f, g: lambda x: f(g(x))
f = lambda x: x+1
g = lambda x: x*2
print(compose(f,g)(3))
```

Output?

- a) 7 ✓
- b) 8
- c) 5
- d) 6

✓ **15.** Composition is different from: * 1/1

- ○ a) Inheritance
- ○ b) Polymorphism
- ○ c) Encapsulation
- ◉ d) All of these ✓

✓ **16.** Composition is often preferred over inheritance because: * 1/1

- ○ a) More flexible
- ○ b) Easier to test
- ○ c) Encourages modularity
- ◉ d) All of the above ✓

✓ **17.** Example of class composition: * 1/1

```
class Engine:
    def start(self): return "Engine started"
class Car:
    def __init__(self):
        self.engine = Engine()
mycar = Car()
print(mycar.engine.start())
```

Output?

- a) Error
- b) Engine started ✓
- c) Car started
- d) None

✓ **18.** In the above example, Car *has-a* Engine. This is: * 1/1

- a) Inheritance
- b) Composition ✓
- c) Polymorphism
- d) Abstraction

✓ **19.** Inheritance is a **"is-a"** relationship, while composition is a: * 1/1

- (●) a) "has-a" relationship ✓
- ( ) b) "does-a" relationship
- ( ) c) "belongs-to" relationship
- ( ) d) None

✓ **20.** Which is more reusable in large applications? * 1/1

- ( ) a) Inheritance
- (●) b) Composition ✓

✓ **21.** Composition allows: * 1/1

- ( ) a) Code duplication
- (●) b) Code reusability ✓
- ( ) c) Less modularity
- ( ) d) Poor maintainability

✓ **22.** Example: * 1/1

```
def add(x): return x+5

def mul(x): return x*2

h = lambda x: add(mul(x))

print(h(3))
```

Output?

- (●) a) 11 ✓
- ( ) b) 16
- ( ) c) 13
- ( ) d) 12

✓ **23.** Nested function composition is also known as: * 1/1

- (●) a) Function chaining ✓
- ( ) b) Function linking
- ( ) c) Function joining
- ( ) d) Function recursion

✓ **24.** Which pattern is often implemented using composition? * 1/1

- ( ) a) Strategy pattern
- ( ) b) Singleton pattern
- ( ) c) Decorator pattern
- (●) d) Both a and c ✓

✓ **25.** Decorators in Python are an example of: * 1/1

- a) Inheritance
- b) Composition ✓
- c) Abstraction
- d) Polymorphism

✓ **26.** Which type of functions are often composed together? * 1/1

- a) Pure functions ✓
- b) Recursive functions
- c) Class methods
- d) Static functions only

✓ **27.** Pure functions are preferred in composition because they: * 1/1

- a) Have no side effects
- b) Always return the same output for same input
- c) Easy to test
- d) All of the above ✓

✓ **28.** Example: * 1/1

```
def upper_case(s): return s.upper()
def exclaim(s): return s + "!"
print(exclaim(upper_case("hello")))
```

Output?

- ○ a) hello!
- ◉ b) HELLO! ✓
- ○ c) Hello!
- ○ d) error

✓ **29.** Function composition often improves: * 1/1

- ○ a) Modularity
- ○ b) Reusability
- ○ c) Readability
- ◉ d) All of the above ✓

✓ **30.** Which is an advantage of class composition? * 1/1

- ○ a) Avoids tight coupling
- ○ b) Allows flexibility
- ○ c) Can reuse existing classes
- ◉ d) All of the above ✓

✓ **31.** Example: * 1/1

```
def f(x): return x-1
def g(x): return x*x
def h(x): return f(g(x))
print(h(5))
```

Output?

- (●) a) 24 ✓
- ( ) b) 25
- ( ) c) 20
- ( ) d) 26

✓ **32.** Composition can be seen as: * 1/1

- (●) a) "Build bigger things out of smaller things" ✓
- ( ) b) "Replace one thing with another"
- ( ) c) "Delete unnecessary code"
- ( ) d) None

✓ **33.** The functools.reduce function can be used for: * 1/1

- (●) a) Function composition over a sequence ✓
- ( ) b) Sorting
- ( ) c) Looping only
- ( ) d) String operations

✓ **34.** Example with map: * 1/1

nums = [1,2,3]

print(list(map(lambda x: (x*2)+1, nums)))

Output?

- a) [2,3,4]
- b) [3,5,7] ✓
- c) [1,2,3]
- d) [2,4,6]

✓ **35.** Composition vs Inheritance: Which promotes loose coupling? * 1/1

- a) Inheritance
- b) Composition ✓

✓ **36.** Function composition is closely related to which paradigm? * 1/1

- a) Object-oriented programming
- b) Functional programming ✓
- c) Procedural programming
- d) Logic programming

✓ **37.** Example: * 1/1

```
def add5(x): return x+5
def mul3(x): return x*3
result = (lambda f,g: lambda x: f(g(x)))(add5,mul3)
print(result(2))
```

Output?

- a) 21
- b) 16
- c) 11 ✓
- d) 17

✓ **38.** Composing f and g gives a new function h(x) = f(g(x)). This is: * 1/1

- a) Mathematical definition of composition ✓
- b) Python only concept
- c) OOP concept only
- d) None

✓ **39.** Using composition helps in: * 1/1

- a) Smaller reusable code blocks
- b) Avoiding duplicate logic
- c) Easier debugging
- d) All of the above ✓

✓ **40.** Which is **NOT** an example of composition? * 1/1

- a) f(g(x))
- b) Car has Engine
- c) Dog is Animal ✓
- d) Using decorator on function

✓ **41.** A class with multiple composed objects is an example of: * 1/1

- a) Aggregation ✓
- b) Polymorphism
- c) Inheritance
- d) Abstraction

✓ **42.** Which keyword is *not* required for composition? * 1/1

- a) class
- b) def
- c) self
- d) super ✓

✓ **43.** Composition encourages: * 1/1

- a) Tight coupling
- b) Loose coupling ✓
- c) No coupling
- d) Dynamic typing only

✓ **44.** Example: * 1/1

```
def capitalize(s): return s.capitalize()

def greet(s): return "Hello " + s

print(greet(capitalize("world")))
```

Output?

- a) hello world
- b) Hello World ✓
- c) Hello world
- d) HELLO WORLD

✓ **45.** Which is closer to composition in OOP? * 1/1

- a) Association
- b) Aggregation
- c) Delegation
- d) All of these ✓

✓ **46.** Composition in functions creates: * 1/1

- (●) a) Pipeline of operations ✓
- ( ) b) Random execution
- ( ) c) Separate threads
- ( ) d) None

✓ **47.** Example: * 1/1

```
def inc(x): return x+1
def square(x): return x*x
def compose(f,g):
    return lambda x: f(g(x))
h = compose(square, inc)
print(h(4))
```

Output?

- ( ) a) 16
- (●) b) 25 ✓
- ( ) c) 20
- ( ) d) 24

✓ **48.** Which operator can simulate composition in functional libraries? * 1/1

- ○ a) >>
- ○ b) <<
- ○ c) |
- ◉ d) All of these (depending on implementation) ✓

✓ **49.** Example: * 1/1

```
class A:
    def hello(self): return "Hello"
class B:
    def __init__(self):
        self.a = A()
obj = B()
print(obj.a.hello())
```

Output?

- ○ a) Error
- ◉ b) Hello ✓
- ○ c) None
- ○ d) World

✓ **50.** The main advantage of composition is: * 1/1

- a) Code duplication
- b) Code reuse and flexibility ✓
- c) More dependencies
- d) Less modularity