

**Vimbai Chatindo H250434Y**

**Principles of Programming Languages**

**Information Technology**

**Assignment**

**1. Compare procedural programming and functional programming, giving one code example for each.**

Procedural programming is based on step-by-step procedures or functions that perform specific tasks. It focuses on how tasks are done using loops, conditionals, and variables.  
Functional programming is based on mathematical functions. It focuses on what to compute rather than how, and it avoids mutable data or state.  
  
 def square\_numbers(nums):  
 squares = []  
 for n in nums:  
 squares.append(n \*\* 2)  
 return squares  
  
 print(square\_numbers([1, 2, 3]))  
  
 print(list(map(lambda x: x \*\* 2, [1, 2, 3])))

**2. Write a recursive function to calculate the factorial of a number n.**

Recursion is when a function calls itself until a base condition is met.  
  
 def factorial(n):  
 if n == 0 or n == 1:  
 return 1  
 else:  
 return n \* factorial(n - 1)  
  
 print(factorial(5))

**3. Explain the purpose of importing modules and write code to import the random module.**

Modules are reusable code libraries that help organize code and avoid repetition.  
  
 import random  
 number = random.randint(1, 10)  
 print("Random number:", number)

**4. Define semantics and show code where syntax is correct but semantics are wrong.**

Syntax defines how code must be written (grammar rules), while semantics defines what the code means.  
  
 age = "twenty"  
 print(age + 5) # Syntax correct but semantically wrong

**5. Write three unit test methods using the unittest module.**

import unittest  
 def add(a, b):  
 return a + b  
  
 class TestMath(unittest.TestCase):  
 def test\_add\_positive(self):  
 self.assertEqual(add(2, 3), 5)  
 def test\_add\_negative(self):  
 self.assertEqual(add(-1, -2), -3)  
 def test\_add\_mixed(self):  
 self.assertEqual(add(-1, 1), 0)  
  
 if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

**6. Explain the evaluation strategy for function arguments.**

Evaluation strategy determines when and how function arguments are evaluated.  
Eager evaluation: Arguments are evaluated before the function executes.  
Lazy evaluation: Arguments are evaluated only when needed.  
Python uses eager evaluation.

**7. Define programming language paradigm and list three major paradigms.**

A programming paradigm is a style or way of programming that defines how code and logic are structured.  
Major paradigms:  
1. Procedural Programming  
2. Object-Oriented Programming  
3. Functional Programming

**8. Write a function that demonstrates the use of closures.**

A closure is a function that remembers variables from its enclosing scope even after that scope has finished executing.  
  
 def outer\_function(x):  
 def inner\_function(y):  
 return x + y  
 return inner\_function  
  
 add\_five = outer\_function(5)  
 print(add\_five(10))

**9. Distinguish between a syntax error and a semantic error.**

Syntax error: Code violates grammar rules and won't run.  
Semantic error: Code runs but gives wrong meaning.  
  
 # Syntax error  
 print("Hello"   
 # Semantic error  
 name = 10  
 print("Hello " + name)

**10. Write a lambda expression to find the square of a number and explain lambda calculus.**

Lambda expressions are anonymous functions.  
  
 square = lambda x: x \*\* 2  
 print(square(4))  
  
Lambda calculus is the mathematical foundation of functional programming.

**11. Define lexical scope and dynamic scope.**

Lexical scope: Determined by where variables are written.  
Dynamic scope: Determined at runtime by call stack.  
  
 x = 10  
 def outer():  
 x = 20  
 def inner():  
 print(x)  
 inner()  
 outer()

**12. List three differences between static typing and dynamic typing.**

Static typing:  
- Type checked at compile time  
- Must declare types  
- Example: Java, C++  
  
Dynamic typing:  
- Type checked at runtime  
- No type declaration needed  
- Example: Python, JavaScript

**13. List the Four Pillars of OOP and demonstrate each.**

1. Encapsulation  
2. Inheritance  
3. Polymorphism  
4. Abstraction  
  
 class Animal:  
 def speak(self):  
 pass  
 class Dog(Animal):  
 def speak(self):  
 return "Woof!"  
 class Cat(Animal):  
 def speak(self):  
 return "Meow!"  
 print(Dog().speak(), Cat().speak())

**14. Write code showing pattern matching.**

value = 3.5  
 match value:  
 case int():  
 print("Integer")  
 case float():  
 print("Float")  
 case str():  
 print("String")  
 case \_:  
 print("Unknown type")

**15. Create classes showing inheritance and overriding.**

class Animal:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def speak(self):  
 return "Some sound"  
 class Dog(Animal):  
 def speak(self):  
 return "Woof!"  
 d = Dog("Buddy")  
 print(d.name, "says", d.speak())

**16. Compare lazy evaluation (generators) vs eager evaluation.**

Eager: computes all results immediately.  
Lazy: computes when needed.  
  
 nums = [n \* 2 for n in range(5)]  
 print(nums)  
 nums\_gen = (n \* 2 for n in range(5))  
 for n in nums\_gen:  
 print(n)

**17. Explain the difference between a compiler and an interpreter.**

Compiler: Translates entire code to machine code first. Example: C++  
Interpreter: Executes code line by line. Example: Python

**18. State three identifier naming rules with valid and invalid examples.**

1. Must start with letter or underscore.  
2. Cannot start with number.  
3. Cannot contain spaces.  
  
Valid: name\_1, \_age  
Invalid: 1name, full-name

**19. Explain the difference between compiled and interpreted languages.**

Compiled: Translated fully before execution. Faster.  
Interpreted: Executed line by line. Slower.  
  
Compiled: C++  
Interpreted: Python

**20. Write code demonstrating first-order functions.**

def greet(name):  
 return "Hello, " + name  
 def call\_func(func, value):  
 return func(value)  
 print(call\_func(greet, "Kai"))

**21. Explain polymorphism in object-oriented programming using class examples.**

Polymorphism allows methods with the same name to behave differently in different classes.  
  
 class Bird:  
 def fly(self):  
 return "Flies high"  
 class Penguin(Bird):  
 def fly(self):  
 return "Cannot fly"  
 for obj in [Bird(), Penguin()]:  
 print(obj.fly())

**22. Define semantics and show code where syntax is correct but semantics are wrong.**

Semantics is the meaning of code.  
  
 x = "5"  
 print(x + 10)

**23. Write three unit test methods using the unittest module.**

import unittest  
 def add(a, b):  
 return a + b  
  
 class TestMath(unittest.TestCase):  
 def test\_add\_positive(self):  
 self.assertEqual(add(2, 3), 5)  
 def test\_add\_negative(self):  
 self.assertEqual(add(-1, -2), -3)  
 def test\_add\_mixed(self):  
 self.assertEqual(add(-1, 1), 0)  
  
 if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

**24. Explain the evaluation strategy for function arguments.**

Python uses eager evaluation: evaluates arguments before calling the function.  
Lazy evaluation delays evaluation until needed.

**25. Compare recursive vs iterative solutions with code examples.**

Recursive uses self-calls, iterative uses loops.  
  
 def factorial\_rec(n):  
 if n == 0:  
 return 1  
 return n \* factorial\_rec(n - 1)  
  
 def factorial\_iter(n):  
 result = 1  
 for i in range(1, n + 1):  
 result \*= i  
 return result

**26. Write a class demonstrating encapsulation.**

class BankAccount:  
 def \_\_init\_\_(self, balance):  
 self.\_\_balance = balance  
 def get\_balance(self):  
 return self.\_\_balance  
 def deposit(self, amount):  
 self.\_\_balance += amount  
 acc = BankAccount(1000)  
 acc.deposit(500)  
 print(acc.get\_balance())

**27. Explain why version control is important for software projects.**

Version control allows tracking code changes, collaboration, and rollback to previous versions (e.g., Git).

**28. Write Git workflow commands.**

git init  
git add \*.py  
git commit -m "Initial commit"  
git remote add origin <repository-URL>  
git push -u origin main

**29. Create a simple DSL example for a specific domain.**

def calc(expression):  
 return eval(expression)  
 print(calc("2 \* (3 + 4)"))

**30. Show how operator precedence works in result = 2 + 3 \* 4 - 1 // 2.**

1. Multiplication  
2. Floor division  
3. Addition/Subtraction  
  
Result: 2 + 12 - 0 = 14

**31. Explain memory management including reference counting.**

Memory management allocates and frees memory automatically.  
Python uses reference counting and garbage collection to manage memory.