Python Programming Notes

1. Simple Calculator

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        return "Cannot divide by zero"
    return x / y

def main():
    while True:
        try:
            print("\n--- Calculator ---")
            print("Select operation:")
            print("1. Add")
            print("2. Subtract")
            print("3. Multiply")
            print("4. Divide")

            choice = input("Enter choice (1/2/3/4): ")

            if choice in ('1', '2', '3', '4'):
                num1 = float(input("Enter first number: "))
                num2 = float(input("Enter second number: "))

                if choice == '1':
                    print(f"{num1} + {num2} = {add(num1, num2)}")

                elif choice == '2':
                    print(f"{num1} - {num2} = {subtract(num1, num2)}")

                elif choice == '3':
                    print(f"{num1} * {num2} = {multiply(num1, num2)}")

                elif choice == '4':
                    print(f"{num1} / {num2} = {divide(num1, num2)}")
            else:
                print("Invalid Input")

        except ValueError:
            print("Error: Please enter valid numbers.")

        # Ask user if they want to continue
```

```
         next_calculation = input("Do you want to perform another calculation? (yes/no):
").lower()
        if next_calculation != 'yes':
            print("Exiting calculator. Goodbye!")
            break

if __name__ == "__main__":
    main()
```

2. Basic While Loops

3. Advanced While Loops

```
import random
import time

def collatz_conjecture(n):
    print(f"\n--- 1. Collatz Conjecture (Starting at {n}) ---")
    steps = 0
    while n != 1:
        print(n, end=" -> ")
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        steps += 1
    print(1)
    print(f"Reached 1 in {steps} steps.")

def newton_sqrt(number, tolerance=1e-7):
    print(f"\n--- 2. Newton's Method for Square Root of {number} ---")
    guess = number / 2.0
    iteration = 0
    while True:
        iteration += 1
        new_guess = (guess + number / guess) / 2.0
        diff = abs(guess - new_guess)
        print(f"Iteration {iteration}: {new_guess:.9f} (diff: {diff:.9f})")
        if diff < tolerance:
            break
        guess = new_guess
    print(f"Approximate square root: {new_guess}")

def prime_factory(count_needed):
    print(f"\n--- 3. Generating first {count_needed} Prime Numbers ---")
    primes = []
    num = 2
    while len(primes) < count_needed:
        # Check if num is prime
        is_prime = True
        divisor = 2
        while divisor * divisor <= num:
            if num % divisor == 0:
                is_prime = False
                break
            divisor += 1

        if is_prime:
            primes.append(num)
        num += 1
    print(f"Primes: {primes}")
```

```
def binary_search_game():
    print("\n--- 4. Computer Guesses Your Number (Binary Search) ---")
    print("Think of a number between 1 and 100.")
    input("Press Enter when you are ready...")

    low = 1
    high = 100
    attempts = 0

    while low <= high:
        attempts += 1
        guess = (low + high) // 2
        print(f"My guess is {guess}.")
        feedback = input("Is it too (h)igh, too (l)ow, or (c)orrect? ").lower()

        if feedback == 'c':
            print(f"I got it in {attempts} attempts!")
            return
        elif feedback == 'h':
            high = guess - 1
        elif feedback == 'l':
            low = guess + 1
        else:
            print("Please enter 'h', 'l', or 'c'.")

    print("I think you might be cheating... I can't guess it!")

def bank_system_simulation():
    print("\n--- 5. Simple Bank System Simulation ---")
    balance = 1000
    while True:
        print("\n--- Bank Menu ---")
        print(f"Current Balance: ${balance}")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Exit")

        choice = input("Select an option: ")

        if choice == '1':
            try:
                amount = float(input("Enter deposit amount: "))
                if amount > 0:
                    balance += amount
                    print(f"Deposited ${amount}. New balance: ${balance}")
                else:
                    print("Invalid amount.")
            except ValueError:
                print("Invalid input.")
```

```
        elif choice == '2':
            try:
                amount = float(input("Enter withdrawal amount: "))
                if 0 < amount <= balance:
                    balance -= amount
                    print(f"Withdrew ${amount}. New balance: ${balance}")
                elif amount > balance:
                    print("Insufficient funds.")
                else:
                    print("Invalid amount.")
            except ValueError:
                print("Invalid input.")

        elif choice == '3':
            print("Exiting Bank System.")
            break
        else:
            print("Invalid choice, please try again.")

def main():
    print("Demonstrating Complex Python While Loops")

    collatz_conjecture(27)
    newton_sqrt(25)
    prime_factory(10)

    # Interactive parts
    # binary_search_game()
    # bank_system_simulation()

if __name__ == "__main__":
    main()
```

4. Function Examples

```
import time
import functools

# --- 1. Decorators (Modifying behavior) ---
def timer_decorator(func):
    """
    A decorator that measures the execution time of a function.
    """
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
          print(f"Function '{func.__name__}' took {end_time - start_time:.6f} seconds to
execute.")
        return result
    return wrapper

@timer_decorator
def slow_function(seconds):
    """
    A function that sleeps to demonstrate the timer decorator.
    """
    print(f"Sleeping for {seconds} seconds...")
    time.sleep(seconds)
    return "Wake up!"

# --- 2. Closures (Function factories) ---
def multiplier_factory(factor):
    """
    Returns a function that multiplies its input by 'factor'.
    This demonstrates closures: the inner function remembers 'factor'.
    """
    def multiplier(number):
        return number * factor
    return multiplier

# --- 3. Generators (Yielding values lazily) ---
def fibonacci_generator(limit):
    """
    A generator that yields Fibonacci numbers up to a limit.
    Memory efficient compared to creating a full list.
    """
    a, b = 0, 1
    while a <= limit:
        yield a
        a, b = b, a + b
```

```
# --- 4. Recursion (Function calling itself) ---
def tower_of_hanoi(n, source, target, auxiliary):
    """
    Solves the Tower of Hanoi puzzle recursively.
    """
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return

    tower_of_hanoi(n - 1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, target, source)

# --- 5. Initial Examples (kept for reference) ---
def greet(name: str) -> str:
    return f"Hello, {name}!"

def power(base, exponent=2):
    return base ** exponent

def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

def sum_all(*args):
    return sum(args)

def build_profile(first, last, **user_info):
    profile = {'first_name': first, 'last_name': last}
    profile.update(user_info)
    return profile

def apply_operation(x, y, operation):
    return operation(x, y)

def main():
    print("--- Advanced Python Function Examples ---\n")

    # 1. Decorators
    print("1. Decorators:")
    slow_function(0.5)

    # 2. Closures
    print("\n2. Closures:")
    double = multiplier_factory(2)
    triple = multiplier_factory(3)
    print(f"   Double of 10: {double(10)}")
    print(f"   Triple of 10: {triple(10)}")

    # 3. Generators
    print("\n3. Generators (Fibonacci up to 20):")
    for num in fibonacci_generator(20):
```

```
        print(num, end=" ")
    print()

    # 4. Recursion
    print("\n4. Recursion (Tower of Hanoi - 3 Disks):")
    tower_of_hanoi(3, 'A', 'C', 'B')

    # Quick run of basic examples
    print("\n--- Basic Examples (Recap) ---")
    print(greet("Alice"))
    print(f"Power: {power(5)}")
    print(f"Sum *args: {sum_all(1, 2, 3)}")

if __name__ == "__main__":
    main()
```

5. Basic Arithmetic (Sum/Sub)

```
#def sum_of_numbers(a, b):
   # print(f"enter first number: {a}")
   # print(f"enter second number: {b}")
   # sum=a+b
   # print(f"the sum of two numbers are: {sum}")

#sum_of_numbers(10,20)
def read_two_numbers():
    a = int(input("enter first number: "))
    b = int(input("enter second number: "))
    return a, b

def substraction_of_two_numbers(a,b):
    sub=a-b
    return sub

def addition_of_two_numbers(a,b):
    add=a+b
    return add


a, b=read_two_numbers()
sum = addition_of_two_numbers(a,b)
sub = substraction_of_two_numbers(a,b)
print(f"the substraction of two numbers are: {sub}")
print(f"the addition of two numbers are: {sum}")
```

6. Multiplication Table

```
def multiplication_table(a, b):
    c = a * b
    return c

if __name__ == "__main__":
    try:
        num = int(input("Enter a number: "))
        for i in range(1, 11):
            print(f"{num} x {i} = {multiplication_table(num, i)}")
    except ValueError:
        print("Please enter a valid integer.")
```

7. Python Data Types

```
def demonstrate_data_types():
    # 1. Numeric Types
    a = 10              # int
    b = 10.5            # float
    c = 1 + 2j          # complex
    print(f"--- Numeric Types ---")
    print(f"{a} is {type(a)}")
    print(f"{b} is {type(b)}")
    print(f"{c} is {type(c)}\n")

    # 2. Sequence Types
    s = "Hello Python"  # str
    l = [1, 2, 3, "Hi"] # list (mutable)
    t = (1, 2, 3, "Hi") # tuple (immutable)
    r = range(5)        # range
    print(f"--- Sequence Types ---")
    print(f"'{s}' is {type(s)}")
    print(f"{l} is {type(l)}")
    print(f"{t} is {type(t)}")
    print(f"{list(r)} is {type(r)}\n")

    # 3. Mapping Type
    d = {"name": "Alice", "age": 25} # dict
    print(f"--- Mapping Type ---")
    print(f"{d} is {type(d)}\n")

    # 4. Set Types
    st = {1, 2, 2, 3}   # set (unordered, unique elements)
    fst = frozenset({1, 2, 3}) # frozenset (immutable set)
    print(f"--- Set Types ---")
    print(f"{st} is {type(st)}")
    print(f"{fst} is {type(fst)}\n")

    # 5. Boolean Type
    is_active = True    # bool
    print(f"--- Boolean Type ---")
    print(f"{is_active} is {type(is_active)}\n")

    # 6. None Type
    empty = None        # NoneType
    print(f"--- None Type ---")
    print(f"{empty} is {type(empty)}")

if __name__ == "__main__":
    demonstrate_data_types()
```

8. Dynamic Typing Demo

```
def demonstrate_dynamic_typing():
    print("--- Python Dynamic Typing Demo ---")

    # 1. No explicit type declaration needed
    # The interpreter infers the type at runtime.
    x = 10
    print(f"Initial: x = {x}, Type: {type(x)}")

    # 2. Variable can change types (reassignment)
    # This is not possible in statically typed languages like C++ or Java.
    x = "Hello, Python!"
    print(f"After reassignment: x = '{x}', Type: {type(x)}")

    # 3. List can hold multiple types
    mixed_list = [1, "Python", 3.14, True]
    print(f"\nMixed List: {mixed_list}")
    for item in mixed_list:
        print(f"  Item: {item}, Type: {type(item)}")

    # 4. Impact: No need to declare types for functions
    def add_anything(a, b):
        return a + b

    print(f"\nFunction 'add_anything':")
    print(f"  Integers: 5 + 5 = {add_anything(5, 5)}")
    print(f"  Strings: 'Hello' + ' World' = {add_anything('Hello', ' World')}")
    print(f"  Lists: [1] + [2] = {add_anything([1], [2])}")

if __name__ == "__main__":
    demonstrate_dynamic_typing()
```