

Python Fundamentals: Practice Worksheet

1. Lists

Lists are ordered, mutable collections that can store elements of different data types.

Key Operations:

- **Creation:** Using square brackets `[]`
- **Indexing:** Accessing elements using zero-based indices
- **Slicing:** Extracting portions of lists
- **Methods:** `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `sort()`, `reverse()`

Examples:

```
python
```

```
# Creating lists
```

```
fruits = ["apple", "banana", "cherry"]
```

```
mixed_list = [1, "hello", 3.14, True]
```

```
# Indexing (positive and negative)
```

```
first_fruit = fruits[0] # "apple"
```

```
last_fruit = fruits[-1] # "cherry"
```

```
# Slicing
```

```
subset = fruits[0:2] # ["apple", "banana"]
```

```
# Common operations
```

```
fruits.append("orange") # Add one element
```

```
fruits.extend(["kiwi", "mango"]) # Add multiple elements
```

```
fruits.insert(1, "pear") # Insert at specific position
```

```
fruits.remove("banana") # Remove by value
```

```
popped_fruit = fruits.pop() # Remove and return last element
```

```
fruits.sort() # Sort in-place
```

```
fruits.reverse() # Reverse in-place
```

```
# List comprehensions
```

```
squares = [x**2 for x in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
even_nums = [x for x in range(20) if x % 2 == 0] # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Practice Assignment 1:

Write a function that takes a list of numbers and returns a new list containing only the even numbers.

python

```
def filter_even(numbers):  
    # Your code here  
    pass  
  
# Test case  
print(filter_even([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])) # Should return [2, 4, 6, 8, 10]
```

2. Functions

Functions are reusable blocks of code designed to perform specific tasks.

Key Concepts:

- **Definition:** Using `def` keyword
- **Parameters:** Input values
- **Return:** Output values
- **Scope:** Variable visibility
- **Default arguments:** Providing fallback values
- **Arbitrary arguments:** Handling variable number of arguments

Examples:

python

Basic function

```
def greet(name):  
    return f"Hello, {name}!"
```

Function with default parameter

```
def greet_with_default(name="Guest"):  
    return f"Hello, {name}!"
```

Function with multiple parameters

```
def add(a, b):  
    return a + b
```

*# Function with *args (variable positional arguments)*

```
def sum_all(*numbers):  
    return sum(numbers)
```

*# Function with **kwargs (variable keyword arguments)*

```
def create_profile(**details):  
    return details
```

Combining parameters

```
def mixed_args(a, b, *args, **kwargs):  
    print(f"a: {a}, b: {b}")  
    print(f"args: {args}")  
    print(f"kwargs: {kwargs}")
```

Examples of calling these functions

```
print(greet("Alice")) # "Hello, Alice!"  
print(greet_with_default()) # "Hello, Guest!"  
print(greet_with_default("Bob")) # "Hello, Bob!"  
print(add(5, 3)) # 8  
print(sum_all(1, 2, 3, 4, 5)) # 15  
print(create_profile(name="Alice", age=25, city="New York")) # {'name': 'Alice', 'age': 25, 'c'  
mixed_args(1, 2, 3, 4, 5, name="Alice", city="New York")
```

Practice Assignment 2:

Write a function that calculates the factorial of a number recursively.

python

```
def factorial(n):  
    # Your code here  
    pass  
  
# Test cases  
print(factorial(5)) # Should return 120  
print(factorial(0)) # Should return 1
```

3. Tuples

Tuples are ordered, immutable collections that can store elements of different data types.

Key Operations:

- **Creation:** Using parentheses `()`
- **Indexing:** Similar to lists
- **Unpacking:** Assigning tuple elements to variables
- **Methods:** `count()`, `index()`

Examples:

python

```
# Creating tuples
coordinates = (10, 20)
person = ("Alice", 25, "New York")
singleton = (42,) # Note the comma is needed for single-element tuples

# Indexing
x = coordinates[0] # 10
name = person[0] # "Alice"

# Tuple unpacking
name, age, city = person
x, y = coordinates

# Tuple methods
colors = ("red", "green", "blue", "red", "yellow")
red_count = colors.count("red") # 2
green_index = colors.index("green") # 1

# Tuples are immutable
# coordinates[0] = 15 # This will raise TypeError

# Multiple return values (actually returns a tuple)
def get_dimensions():
    return 1920, 1080

width, height = get_dimensions()
```

Practice Assignment 3:

Write a function that takes a list of tuples, each containing a student's name and score, and returns a list of names of students who scored above 70.

python

```
def high_scorers(student_scores):
    # Your code here
    pass

# Test case
students = [("Alice", 85), ("Bob", 65), ("Charlie", 90), ("David", 70)]
print(high_scorers(students)) # Should return ["Alice", "Charlie"]
```

4. Lambda Functions

Lambda functions are small, anonymous functions defined using the `lambda` keyword.

Key Concepts:

- **Syntax:** `lambda arguments: expression`
- **Use Cases:** Short operations, functions as arguments

Examples:

```
python
```

```
# Basic Lambda function
```

```
square = lambda x: x ** 2
```

```
print(square(5)) # 25
```

```
# Lambda with multiple arguments
```

```
add = lambda a, b: a + b
```

```
print(add(3, 4)) # 7
```

```
# Lambda with conditionals
```

```
is_even = lambda x: True if x % 2 == 0 else False
```

```
print(is_even(4)) # True
```

```
# Lambda with lists
```

```
numbers = [1, 2, 3, 4, 5]
```

```
squared = list(map(lambda x: x**2, numbers)) # [1, 4, 9, 16, 25]
```

```
# Lambda with filter
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
evens = list(filter(lambda x: x % 2 == 0, numbers)) # [2, 4, 6, 8, 10]
```

```
# Lambda with sorted
```

```
students = [("Alice", 85), ("Bob", 65), ("Charlie", 90)]
```

```
sorted_by_score = sorted(students, key=lambda x: x[1], reverse=True)
```

```
# [("Charlie", 90), ("Alice", 85), ("Bob", 65)]
```

Practice Assignment 4:

Use lambda functions with map and filter to create a list of squares of even numbers from a given list.

python

```
def even_squares(numbers):  
    # Your code here  
    pass
```

Test case

```
print(even_squares([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])) # Should return [4, 16, 36, 64, 100]
```

5. While Loops

While loops execute a block of code as long as a condition is true.

Key Concepts:

- **Syntax:** `while condition: block`
- **Control:** break, continue, else

Examples:

python

Basic while loop

```
count = 0
while count < 5:
    print(count)
    count += 1
# Prints: 0, 1, 2, 3, 4
```

While loop with break

```
count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break
# Prints: 0, 1, 2, 3, 4
```

While loop with continue

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
# Prints: 1, 3, 5, 7, 9
```

While loop with else

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("Count reached 5")
# Prints: 0, 1, 2, 3, 4, "Count reached 5"
```

Input validation with while

```
def get_positive_number():
    while True:
        try:
            num = float(input("Enter a positive number: "))
            if num > 0:
                return num
            print("The number must be positive.")
```

```
except ValueError:  
    print("Invalid input. Please enter a number.")
```

Practice Assignment 5:

Write a program that uses a while loop to find the first 10 Fibonacci numbers.

python

```
def first_n_fibonacci(n):  
    # Your code here  
    pass  
  
# Test case  
print(first_n_fibonacci(10)) # Should return [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

6. For Loops

For loops iterate over a sequence (like a list, tuple, string, or range).

Key Concepts:

- **Syntax:** `for item in sequence: block`
- **Control:** break, continue, else
- **Iteration:** range(), enumerate()

Examples:

python

```
# Basic for loop with a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Prints: "apple", "banana", "cherry"

# Loop with range
for i in range(5):
    print(i)
# Prints: 0, 1, 2, 3, 4

# Range with start, stop, step
for i in range(2, 10, 2):
    print(i)
# Prints: 2, 4, 6, 8

# Loop with enumerate
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# Prints: "0: apple", "1: banana", "2: cherry"

# Loop with break
for i in range(10):
    if i == 5:
        break
    print(i)
# Prints: 0, 1, 2, 3, 4

# Loop with continue
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
# Prints: 1, 3, 5, 7, 9

# Loop with else
for i in range(5):
    print(i)
else:
    print("Loop completed")
# Prints: 0, 1, 2, 3, 4, "Loop completed"
```

```
# Nested Loops
for i in range(3):
    for j in range(2):
        print(f"({i}, {j})")
# Prints: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1)
```


Practice Assignment 6:

Write a function that takes a list of numbers and returns the sum of squares of all odd numbers in the list using a for loop.

```
python

def sum_of_odd_squares(numbers):
    # Your code here
    pass

# Test case
print(sum_of_odd_squares([1, 2, 3, 4, 5])) # Should return 35 (1^2 + 3^2 + 5^2 = 1 + 9 + 25 =
```



Challenging Problems

Problem 1: List Manipulation

Write a function that takes a list of integers and returns a new list where each element is the product of all numbers in the original list except the number at that index. Do this without using division.

```
python

def products_except_self(nums):
    # Your code here
    pass

# Test case
print(products_except_self([1, 2, 3, 4])) # Should return [24, 12, 8, 6]
```

Problem 2: Function Decorator

Create a decorator function that measures and prints the execution time of the decorated function.

python

```
import time

def timer_decorator(func):
    # Your code here
    pass

@timer_decorator
def slow_function(n):
    """A deliberately slow function for testing"""
    total = 0
    for i in range(n):
        for j in range(1000000):
            total += i * j
    return total

# Test case
slow_function(5) # Should print the execution time
```

Problem 3: Tuple Processing

Write a function that takes a list of tuples, where each tuple contains a name and an arbitrary number of scores. Return a dictionary that maps each name to their average score.

python

```
def calculate_averages(students_scores):
    # Your code here
    pass

# Test case
scores = [
    ("Alice", 85, 90, 92),
    ("Bob", 70, 65, 80, 85),
    ("Charlie", 95, 92),
    ("David",)
]
print(calculate_averages(scores))
# Should return {'Alice': 89.0, 'Bob': 75.0, 'Charlie': 93.5, 'David': 0}
```

Problem 4: Lambda with Reduce

Use lambda functions with reduce to find the maximum element in a list of dictionaries based on a specified key.

```
python

from functools import reduce

def find_max_by_key(items, key):
    # Your code here
    pass

# Test case
students = [
    {"name": "Alice", "score": 85},
    {"name": "Bob", "score": 65},
    {"name": "Charlie", "score": 90},
    {"name": "David", "score": 70}
]
print(find_max_by_key(students, "score")) # Should return {"name": "Charlie", "score": 90}
```

Problem 5: Complex While Loop Logic

Implement a function that simulates the Collatz conjecture: if n is even, divide it by 2; if n is odd, multiply by 3 and add 1. The function should return the number of steps needed to reach 1.

```
python

def collatz_steps(n):
    # Your code here
    pass

# Test cases
print(collatz_steps(1)) # Should return 0
print(collatz_steps(6)) # Should return 8
print(collatz_steps(27)) # Should return 111
```

Problem 6: Nested For Loop Challenge

Write a function that takes a number n and returns a list of all pairs (a, b) where a and b are positive integers, $a \leq b$, and $a + b \leq n$.

python

```
def find_pairs_sum_less_equal(n):  
    # Your code here  
    pass  
  
# Test case  
print(find_pairs_sum_less_equal(5))  
# Should return [(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (3, 1), (3, 2)]
```

Bonus: Combining Everything

Create a program that processes student data using all the concepts we've covered:

python

```
def process_student_data(raw_data):  
    # Your code here  
    pass
```

Test with sample data

```
raw_data = """
```

```
Alice,92,95,88
```

```
Bob,72,65,70
```

```
Charlie,95,98,99
```

```
David,60,58,62
```

```
Eva,88,87,90
```

```
"""
```

Expected output format:

```
"""
```

```
Student Averages:
```

```
Charlie: 97.3 - Grade: A
```

```
Alice: 91.7 - Grade: A
```

```
Eva: 88.3 - Grade: B
```

```
Bob: 69.0 - Grade: D
```

```
David: 60.0 - Grade: D
```

```
Grade Distribution:
```

```
A: 2 students
```




```
B: 1 students
```

```
D: 2 students
```




```
"""
```


Best Practices and Common Pitfalls



Lists:

-  Use list comprehensions for cleaner code
-  Avoid `list.append()` in a loop when you can use a list comprehension
-  Don't use `[]*n` to create lists of lists (creates shallow copies)



Functions:

-  Follow the Single Responsibility Principle
-  Use docstrings to document your functions
-  Avoid mutable default arguments `(def func(a=[]): ...)`



Tuples:

-  Use tuples for immutable data or when returning multiple values
-  Don't try to modify tuple elements




Lambda Functions:

-  Use lambda for simple, one-line functions
-  Avoid complex logic in lambda functions

While Loops:

-  Always ensure the loop condition will eventually become False
-  Avoid infinite loops without break conditions

For Loops:

-  Use `enumerate()` when you need both index and value
-  Use `zip()` to iterate over multiple sequences together
-  Avoid modifying the sequence you're iterating over