PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

1. What is the difference between a function and a method in Python?

In Python, the terms **function** and **method** refer to callable objects that perform specific tasks, but they have distinct contexts and usage.

1. **Function**:

- A **function** is a block of code that is designed to perform a specific task. It is defined using the `def` keyword.
- Functions can be called on their own, independent of any object.
- Functions can take any number of arguments and can return a value (or not).

Example of a function:

python

Copy code

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

result = greet("Alice")
print(result)  # Output: Hello, Alice!
```

2. **Method**:

- A **method** is a function that is associated with an object and is defined within a class.
- Methods are invoked on an instance of a class (an object) or the class itself.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

- The first argument of a method is typically self, which refers to the instance of the object (for instance methods), or cls, which refers to the class itself (for class methods).

Example of a method:

```
python
Copy code
class Person:
    def __init__(self, name):
        self.name = name

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


# Create an instance of Person
person = Person("Alice")
result = person.greet()  # Calling the method on an
instance
print(result)  # Output: Hello, Alice!
```

2. Explain the concept of function arguments and parameters in Python.

In Python, functions allow you to bundle code into reusable blocks. When you define a function, you can specify inputs that it needs to work with. These inputs are called **parameters** when defining the function, and **arguments** when you call the function.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

1. Parameters:

Parameters are the names you use in a function definition to represent the data that will be passed to the function. They are placeholders for the values that the function will operate on.

Example:

```
python
Copy code
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")
```

In this example:

- name and age are **parameters** of the function greet.
- They act as variables within the function definition that will hold the values provided when the function is called.

2. Arguments:

Arguments are the actual values that you pass into a function when you call it. These values correspond to the parameters defined in the function.

Example of calling the function:

```
python
Copy code
greet("Alice", 30)
```

In this call:

- `"Alice"` is the **argument** passed for the `name` parameter.
- `30` is the **argument** passed for the `age` parameter.

Key Points:

- **Parameters** are defined in the function signature (the definition of the function).
- **Arguments** are the actual values you provide when calling the function.

3. What are the different ways to define and call a function in Python?

In Python, functions can be defined and called in several different ways. Here are the key methods:

1. Defining and Calling a Function using `def`

This is the most common and standard way to define and call a function in Python.

Syntax:

python
Copy code
```
def function_name(parameters):
    # function body
    return result
```

Example:

python
Copy code

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

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

# Calling the function
print(greet("Alice"))
```

2. Using Lambda Functions (Anonymous Functions)

Lambda functions are small, anonymous functions defined using the lambda keyword. These are often used when you need a short, throwaway function and don't want to define a full function with def.

Syntax:

python

Copy code

```
lambda parameters: expression
```

Example:

python

Copy code

```
greet = lambda name: f"Hello, {name}!"
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

Calling the function

print(greet("Bob"))

3. Function Calling with Arguments

Functions can be called in a variety of ways based on how you pass the arguments:

Positional Arguments: Arguments are passed in the order defined.

Keyword Arguments: Arguments are passed by explicitly naming the parameters.

Default Arguments: Parameters can have default values if no argument is provided.

- **Variable-Length Arguments**: You can pass an arbitrary number of arguments using *args (for positional arguments) or ****kwargs** (for keyword arguments).

Example 1: Positional Arguments

python

Copy code

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


print(add(3, 5))  # Output: 8
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

Example 2: Keyword Arguments

python

Copy code

```
def greet(name, age):
    return f"Hello, {name}. You are {age} years old."


print(greet(name="Alice", age=30))  # Output: Hello, Alice. You are 30 years old.
```

Example 3: Default Arguments

python

Copy code

```
def greet(name, age=25):
    return f"Hello, {name}. You are {age} years old."


print(greet("Bob"))  # Output: Hello, Bob. You are 25 years old.
```

Example 4: Variable-Length Arguments (*args and **kwargs)

python

Copy code

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

```
def greet(*args, **kwargs):
    print(f"Arguments: {args}")
    print(f"Keyword Arguments: {kwargs}")


greet(1, 2, 3, name="Alice", age=30)
```

Output:

css

Copy code

```
Arguments: (1, 2, 3)
Keyword Arguments: {'name': 'Alice', 'age': 30}
```

4. Calling a Function Using globals() or locals()

Functions can also be called dynamically using the globals() or locals() functions, which allow access to variables and functions in the global or local scope.

Example using globals():

python

Copy code

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

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


func_name = "greet"
print(globals()[func_name]("Charlie"))  # Output: Hello, Charlie!
```

5. **Calling Functions Using** functools.partial

functools.partial is used to create a new function with some arguments fixed. It allows partial application of arguments to a function.

Example:

python

Copy code

```
from functools import partial


def multiply(a, b):
    return a * b


# Create a new function where a is always 2
double = partial(multiply, 2)
```

print(double(5)) # Output: 10

6. Calling Functions as Objects (First-Class Functions)

In Python, functions are first-class objects, meaning they can be assigned to variables, passed around, and called like any other object.

Example:

python

Copy code

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


# Assign function to a variable
greeting_function = greet


# Call function through the variable
print(greeting_function("Dave"))  # Output: Hello, Dave!
```

7. Method Calls in Object-Oriented Programming

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

In OOP, methods are functions associated with objects (instances of classes). These methods are called using the object or class.

Example:

python

Copy code

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, {self.name}!"
# Create an object
person = Person("Eva")

# Calling a method on the object
print(person.greet())  # Output: Hello, Eva!
```

8. Function Wrapping (Decorators)

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

Decorators are functions that modify the behavior of other functions. They are applied using the @ symbol.

Example:

python

Copy code

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper


@decorator
def say_hello():
    print("Hello!")


say_hello()
```

Output:

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

r

Copy code

Before function call

Hello!

After function call

4. What is the purpose of the `return` statement in a Python function?

The return statement in a Python function is used to exit the function and optionally send a value back to the caller. When a function executes a return statement, it stops executing further code in that function and passes the specified value (or None if no value is provided) back to the place where the function was called.

Here's a breakdown of its purpose:

1. **Exiting the function**: The return statement causes the function to terminate immediately, no matter where it appears in the function.

2. **Returning a value**: The value following the return keyword is sent back to the caller. This allows the function to produce a result that can be used elsewhere in your program.

Example:

python
Copy code

```
def add(a, b):
    return a + b  # returns the sum of a and b

result = add(3, 5)
print(result)  # Output: 8
```

In this example:

- The function add calculates the sum of a and b, and then return a + b sends that result (8) back to the caller.
- The value 8 is then stored in the variable result and printed.

If no return statement is used in a function, it returns None by default.

5. What are iterators in Python and how do they differ from iterables?

- In Python, iterators and iterables are concepts related to the process of looping over data, but they have distinct roles.

1. Iterable:

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

- An **iterable** is any object in Python that can return an iterator. These are objects that can be iterated (looped) over, such as lists, tuples, strings, dictionaries, sets, etc. An iterable must implement the __iter__() method, which returns an iterator.

Example of an iterable:

```
python
Copy code
my_list = [1, 2, 3, 4]
for item in my_list:
    print(item)
```

2. Iterator:

An **iterator** is an object that represents a stream of data and keeps track of the state of iteration. An iterator is an object that implements two methods:

- __iter__(): Returns the iterator object itself. This is required for an object to be an iterator.
- __next__(): Returns the next item from the container. If there are no more items, it raises a StopIteration exception.

Example of an iterator:

```
python
Copy code
my_iter = iter(my_list)  # Get an iterator from the
list

print(next(my_iter))  # Outputs: 1
print(next(my_iter))  # Outputs: 2
```

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

```
print(next(my_iter))  # Outputs: 3
print(next(my_iter))  # Outputs: 4
# next(my_iter) would now raise StopIteration because
there are no more items.
```

Key Differences:

Feature	Iterable	Iterator
Definition	Any object that can return an iterator	An object that performs the iteration
Methods	Must implement `__iter__()` to return an iterator	Must implement both `__iter__()` and `__next__()`
Usage	Can be looped over using `for` loops or passed to `iter()`	Used to iterate one element at a time using `next()`
State	Does not keep track of the iteration state	Keeps track of the current position during iteration
Examples	List, Tuple, String, Dictionary, Set	ListIterator, TupleIterator, StringIterator, etc.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

6. Explain the concept of generators in Python and how they are defined.

Generators in Python

- A **generator** in Python is a special type of iterator that allows you to iterate over a sequence of values, but instead of creating and storing the entire sequence in memory, it generates values on the fly, one at a time, as needed. This makes generators more memory-efficient when working with large datasets or infinite sequences.

Defining a Generator

Generators are defined in Python in two main ways:

1. **Using a Function with** yield**:**
   - A generator function is a regular function but uses the yield keyword instead of return to return values.
   - Each time yield is called, the function's state is saved, and the value is returned to the caller. The next time the generator is called, it resumes from where it left off.

Example:

```
python
Copy code
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

In the above example, **countdown** is a generator function. It starts with n, yields a value, and then decrements n until it reaches 0.

To use the generator:

python

Copy code

```
gen = countdown(5)

for number in gen:

    print(number)
```

Output:

```
Copy code
5
4
3
2
1
```

2. Using a Generator Expression:

- Generator expressions are a more concise way to create generators without defining a separate function.
- They are similar to list comprehensions, but instead of creating a list, they return a generator object.

Example:

```
python
Copy code
gen = (x * x for x in range(5))
for num in gen:
    print(num)
```

Output:

```
Copy code
0
1
4
9
16
```

7. What are the advantages of using generators over regular functions?

Generators offer several advantages over regular functions, especially when working with large datasets, streams of data, or when you need to implement stateful iteration. Here are the key advantages:

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

1. Lazy Evaluation (Memory Efficiency)

- **Generators:** produce values one at a time as needed, instead of computing and returning all values at once (like a regular function that returns a list or other collection). This "lazy" evaluation means that a generator only consumes memory for one item at a time.
- **Advantage:** When working with large datasets, or infinite sequences, this can drastically reduce memory usage compared to storing all values in a list or other collection. For example, when processing a huge file line by line, a generator would load just one line into memory at a time.

2. State Retention Between Iterations

- A generator maintains its state between calls, meaning it "remembers" where it left off, which is useful for complex iteration patterns or stateful computations.
- **Advantage**: You don't need to manually manage state between function calls; the generator handles it for you. This can simplify the code, especially in cases of long-running loops, recursive functions, or sequences where the next value depends on previous computations.

3. Performance Benefits

- Since a generator yields values one at a time, it doesn't need to process the entire dataset in memory at once. This can lead to significant performance improvements, particularly for large data or when the entire dataset isn't needed right away.

- **Advantage**: In cases where the entire result set is not required all at once (e.g., finding the first match in a list, reading a file line by line), you save both memory and processing time.

4. Readable and Expressive Code

- Generators can simplify code that would otherwise require manually managing iteration, such as in cases of complex loops or recursive algorithms.
- **Advantage**: The use of yield can make code more concise, readable, and maintainable compared to managing iteration with explicit states or temporary lists.

5. Easy to Implement Infinite Sequences

- Generators are ideal for working with potentially infinite sequences, like generating Fibonacci numbers or iterating over an endless data stream.
- **Advantage**: You can create "infinite" sequences without worrying about memory consumption, as only the current value is ever computed and held in memory.

6. Control Flow via yield

- Generators can be paused and resumed at any point in their execution using yield. This gives fine control over the flow of execution, allowing you to pause and resume function execution while maintaining context.
- **Advantage**: This can be useful for tasks like coroutines, asynchronous programming, or implementing iterators where you want control over when the function "suspends" and "resumes."

7. Improved Concurrency (with async/await)

- Generators can be used in conjunction with asynchronous programming patterns (e.g., using async def and await) to handle asynchronous operations in a more natural and readable way.
- **Advantage**: In cases where you're working with I/O-bound operations or tasks that involve waiting (e.g., HTTP requests, file reads), asynchronous generators (async def with yield) can help keep code responsive without blocking.

Example: Comparison

Regular Function

```
python
Copy code
def get_numbers():
    return [1, 2, 3, 4, 5]
```

Generator Function

```
python
Copy code
def get_numbers():
    for i in range(1, 6):
        yield i
```

8. What is a lambda function in Python and when is it typically used?

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

A **lambda function** in Python is a small, anonymous function defined with the `lambda` keyword, as opposed to the standard `def` keyword. The general syntax for a lambda function is:

```
python
Copy code
lambda arguments: expression
```

- **`arguments`**: A comma-separated list of parameters (similar to the parameters in a regular function).
- **`expression`**: A single expression that is evaluated and returned when the function is called.

Example of a lambda function:

```
python
Copy code
# A lambda function that adds two numbers
add = lambda x, y: x + y

# Using the lambda function
result = add(5, 3)  # result is 8
```

When is a lambda function typically used?

1. **For short, simple operations**: Lambda functions are useful for cases where you need a small function for a short period of time and don't want to define a full function using `def`. They are typically used for simple expressions.
2. **In functional programming contexts**:
   - They are often passed as arguments to higher-order functions like `map()`, `filter()`, and `reduce()`.

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

- Example with `map()`:

  ```
  python
  Copy code
  numbers = [1, 2, 3, 4]
  squared = list(map(lambda x: x**2, numbers))
  print(squared)  # Output: [1, 4, 9, 16]
  ```

3. **In sorting operations**: Lambda functions are frequently used to define custom sorting criteria.
   - Example with `sorted()`:

     ```
     python
     Copy code
     points = [(1, 2), (3, 1), (5, 4)]
     sorted_points = sorted(points, key=lambda x: x[1])
     print(sorted_points)  # Output: [(3, 1), (1, 2), (5, 4)]
     ```

4. **When defining callbacks**: In event-driven programming or GUI libraries, lambda functions can be used as quick, on-the-fly callback functions.

9. Explain the purpose and usage of the `map()` function in Python.

The `map()` function in Python is used to apply a given function to each item in an iterable (like a list, tuple, etc.) and return a map object (which is an iterator) that yields the results.

Purpose

The main purpose of map() is to transform or modify each item in an iterable using a specified function. It allows you to avoid writing explicit

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

loops and makes your code more concise and readable when you want to apply the same operation to every element in an iterable.

Syntax

python

Copy code

map(function, iterable, ...)

- **function**: A function that will be applied to every item in the iterable(s).
- **iterable**: One or more iterables (e.g., lists, tuples, etc.) to which the function will be applied.

The function should accept as many arguments as there are iterables. If you provide multiple iterables, map() will apply the function in parallel, passing one item from each iterable to the function at a time.

Usage Example

1. Applying a single function to an iterable:

```
python
Copy code
# Example 1: Applying a function to each element of a list
numbers = [1, 2, 3, 4, 5]

# A simple function to square numbers
def square(x):
    return x * x
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

```
squared_numbers = map(square, numbers)

# Convert the result to a list to display the output
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

2. *Using `lambda` functions*:

You can also pass a `lambda` function (anonymous function) to `map()` for simpler code.

```
python
Copy code
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x * x, numbers)
print(list(squared_numbers))  # Output: [1, 4, 9, 16,
25]
```

3. *Applying a function to multiple iterables*:

If you pass more than one iterable to `map()`, the function will receive one item from each iterable in parallel.

```
python
Copy code
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

# A function to add corresponding elements
def add(x, y):
    return x + y

sum_numbers = map(add, numbers1, numbers2)
print(list(sum_numbers))  # Output: [5, 7, 9]
```

4. *Using `map()` with `str` functions:*

You can also use `map()` to apply built-in functions like `str`, `int`, etc.

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

```
python
Copy code
string_numbers = ['1', '2', '3']
integer_numbers = map(int, string_numbers)
print(list(integer_numbers))  # Output: [1, 2, 3]
```

10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

In Python, the `map()`, `reduce()`, and `filter()` functions are all built-in higher-order functions used for functional programming. They allow you to apply operations on collections like lists or iterators in a concise way. Here's a breakdown of the differences between them:

1. `map()` function

- **Purpose**: Applies a function to every item in an iterable (like a list or a tuple) and returns an iterable (specifically, a `map` object, which is an iterator) of the results.
- **Syntax**: `map(function, iterable, ...)`
  - `function`: A function that takes one or more arguments.
  - `iterable`: An iterable like a list, tuple, etc. The function is applied to each item in the iterable.
  - You can pass more than one iterable, and the function will be applied to the items of the iterables in parallel (the function must accept as many arguments as there are iterables).
- **Use case**: Use `map()` when you need to transform or modify each item in an iterable.

  #### Example:

  ```
  python
  ```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

```
Copy code
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

2. `reduce()` function (from the `functools` module)

- **Purpose**: Applies a binary function (a function that takes two arguments) cumulatively to the items in an iterable, reducing the iterable to a single value.
- **Syntax**: `reduce(function, iterable, [initializer])`
  - `function`: A function that takes two arguments (the accumulator and the current item) and returns a result.
  - `iterable`: An iterable to process.
  - `initializer`: (Optional) A starting value for the accumulation. If not provided, the first item of the iterable is used.
- **Use case**: Use `reduce()` when you want to combine the items in an iterable into a single result, such as summing all numbers or multiplying them.

Example:

```
python
Copy code
from functools import reduce

numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24 (1 * 2 * 3 * 4)
```

3. `filter()` function

- **Purpose**: Filters an iterable by applying a function to each item and keeping only those that evaluate to `True` (the function returns `True` or `False`).
- **Syntax**: `filter(function, iterable)`
  - `function`: A function that returns a boolean value (True or False).

PW SKILL
DATA FUNCTIONS ASSIGNENT (THEORY)

  - `iterable`: An iterable to filter.
- **Use case**: Use `filter()` when you need to remove items from an iterable based on a condition.

  Example:

```
python
Copy code
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers)) # Output: [2, 4, 6]
```

11.Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list: [47,11,42,13];

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

11. Ans:- To understand the internal mechanism of the reduce() function in python when performing a sum operation on the given list [47, 11, 42, 13], Steps are given as:-

Given List :-

```
Numbers = [47, 11, 42, 13]
```

Using reduce():

```
from functools import reduce

numbers = [47, 11, 42, 13]
result = reduce(lambda x, y: x + y, numbers)
print(result)
```

The reduce() function from the functools module applies a binary function (in this case, lambda x, y: x + y) cumulatively to the items of the iterable (the list [47, 11, 42, 13], reducing the iterable to a single value.

How it works:

1. First, we have the list:

[47, 11, 42, 13]

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

2. reduce () applies the function lambda x,y: x+y cumulatively:

- First iteration:
- The first two elements of the list, 47 and 11, are passed to the lambda function:

$$x = 47, \quad y = 11$$
$$result = x + y = 47 + 11 = 58$$

- So, after the first iteration, the intermediate result is 58.
- Second iteration:
- Now, the result from the first iteration (58) is combined with the next element (42):

$$x = 58, \quad y = 42$$
$$result = x + y = 58 + 42 = 100$$

- After the second iteration, the intermediate result is 100.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

- Third iteration:

- Next, the result from the second iteration (100) is combined with the last element (13):

$$\boxed{\begin{array}{l} x = 100, \; y = 13 \\ result = x + y = 100 + 13 = 113 \end{array}}$$

- After the third iteration, the final result is 113.

Final Result :

After all the iterations, the sum of all the numbers in the list is 113.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(THEORY)

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.?

Certainly! Below is a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list:

```
python
def sum_of_even_numbers(numbers):
    # Initialize the sum to 0
    total_sum = 0

    # Iterate through each number in the list
    for num in numbers:
        # Check if the number is even
        if num % 2 == 0:
            total_sum += num  # Add even number to the
total sum

    return total_sum
```

Example Usage:

```
python
numbers = [1, 2, 3, 4, 5, 6]
result = sum_of_even_numbers(numbers)
print(result)  # Output will be 12 (2 + 4 + 6)
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

2. Create a Python function that accepts a string and returns the reverse of that string.

You can create a Python function to reverse a string using slicing. Here's an example of how you could implement it:

```python
def reverse_string(input_string):

    # Reverse the string using slicing

    return input_string[::-1]


# Example usage

print(reverse_string("Hello, World!"))  # Output: "!dlroW ,olleH"
```

3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.

Here's a simple Python function that takes a list of integers and returns a new list containing the squares of each number:

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

```
python
def square_numbers(nums):
    return [num ** 2 for num in nums]
```

Example Usage:

```
python
numbers = [1, 2, 3, 4, 5]
squared_numbers = square_numbers(numbers)
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

This function uses list comprehension to iterate over the input list `nums` and squares each number (`num ** 2`), returning a new list with the squared values.

4 . Write a Python function that checks if a given number is prime or not from 1 to 200.

To check if a given number is prime or not within the range of 1 to 200, you can write a Python function that implements the basic logic of prime number checking. A prime number is a number greater than 1 that has no positive divisors other than 1 and itself.

Here’s a Python function that checks if a number is prime:

```
python

def is_prime(n):

    # Prime numbers are greater than 1

    if n <= 1:
```

```
        return False

    # Check divisibility from 2 to sqrt(n)

    for i in range(2, int(n**0.5) + 1):

        if n % i == 0:

            return False

    return True


# Testing the function with numbers from 1 to 200

for number in range(1, 201):

    if is_prime(number):

        print(number)
```

Explanation:

is_prime(n): This function returns True if the number n is prime, and False otherwise.

Edge case check: If the number is less than or equal to 1, it cannot be prime, so we return False.

Efficient checking: Instead of checking all numbers from 2 to n-1, the function only checks up to the square root of n. This is a known optimization because if n has a divisor greater than its square root, the corresponding divisor will be smaller than the square root.

PW SKILL
DATA FUNCTIONS ASSIGNENT (PRACTICAL)

Prime number testing: It then loops through the numbers from 1 to 200, calling is prime for each and printing the prime numbers.

Example Output:

2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

PW SKILL
DATA FUNCTIONS ASSIGNENT (PRACTICAL)

61

67

71

73

79

83

89

97

101

103

107

109

113

127

131

137

139

149

151

157

163

167

173

179

181

191

193

197

199

This will print all prime numbers from 1 to 200.

5 Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms.

To create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms, we can define a class that implements the iterator protocol. This protocol requires the implementation of two methods:

1. __iter__(self): This method returns the iterator object itself (which is typically self).

PW SKILL
DATA FUNCTIONS ASSIGNENT (PRACTICAL)

2. __next__(self): This method returns the next value in the sequence. If there are no more values to return, it should raise a StopIteration exception.

Here is a Python implementation of such an iterator class for the Fibonacci sequence:

```
python
class FibonacciIterator:
    def __init__(self, n):
        """
        Initialize the Fibonacci iterator.

        :param n: The number of terms in the Fibonacci sequence
to generate.
        """
        self.n = n # The number of terms to generate
        self.a, self.b = 0, 1 # The first two numbers in the
Fibonacci sequence
        self.count = 0 # Counter to keep track of how many terms
we've yielded

    def __iter__(self):
        """
        Return the iterator object itself.
        """
        return self

    def __next__(self):
        """
        Return the next Fibonacci number in the sequence.

        Raises:
            StopIteration: If the sequence has been exhausted.
        """
        if self.count >= self.n:
            raise StopIteration  # Stop iteration if we have
generated n terms

        # Calculate the next Fibonacci number
```

```
        fib_number = self.a
        self.a, self.b = self.b, self.a + self.b  # Update the 
next two Fibonacci numbers
        self.count += 1
        return fib_number

# Example usage:
n_terms = 10
fibonacci = FibonacciIterator(n_terms)

for number in fibonacci:
    print(number)
```

Explanation:

1. **Constructor (`__init__`)**:
   - `n` is the number of terms to generate in the Fibonacci sequence.
   - We initialize the first two numbers of the Fibonacci sequence (`a` and `b`) to 0 and 1, respectively.
   - `count` keeps track of how many Fibonacci numbers have been generated.
2. **`__iter__` method**:
   - This method simply returns the iterator object (which is the current instance of the class).
3. **`__next__` method**:
   - This method generates the next Fibonacci number by returning `a`, then updates `a` and `b` to the next two numbers in the sequence (`a` becomes `b`, and `b` becomes the sum of `a` and `b`).
   - If the `count` exceeds or equals `n`, it raises a `StopIteration` to signal that the sequence is finished.

Example Output (for `n terms = 10`):

```
0
1
1
```

```
2
3
5
8
13
21
34
```

This code will print the first 10 terms of the Fibonacci sequence. You can adjust `n terms` to generate a different number of terms.

6. Write a generator function in Python that yields the powers of 2 up to a given exponent.

Here's a Python generator function that yields the powers of 2 up to a given exponent:

```
python
def powers_of_2(exponent):
    for i in range(exponent + 1):
        yield 2 ** i

# Example usage:
for power in powers_of_2(5):
    print(power)
```

Explanation:

- The function `powers_of_2` takes an `exponent` as input.
- It iterates over a range from `0` to `exponent` (inclusive).
- On each iteration, it calculates `2 ** i` (2 raised to the power of `i`) and yields that value.
- The generator will stop once it has yielded values up to `2 ** exponent`.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

Example Output for `powers_of_2(5)`:

```
1
2
4
8
16
32
```

7. Implement a generator function that reads a file line by line and yields each line as a string.

You can implement a generator function in Python to read a file line by line using the yield keyword. This allows you to process each line of the file one at a time, which can be more memory-efficient than reading the entire file into memory.

Here's an example implementation:

```
python
def read_file_line_by_line(file_path):
    """Generator function that reads a file line by line."""
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()  # Remove the trailing newline
character from each line
```

Explanation:

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

- `open(file_path, 'r')`: Opens the file in read mode.
- `for line in file`: Iterates over each line in the file.
- `yield line.strip()`: Yields each line with the trailing newline removed (`strip()` removes the newline character at the end of each line).

Example Usage:

```
python
file_path = 'your_file.txt'

for line in read_file_line_by_line(file_path):
    print(line)
```

This will print each line of the file without storing the entire file content in memory, which is useful when working with large files.

8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple.

To sort a list of tuples based on the second element of each tuple using a lambda function in Python, you can use the sorted() function and specify the sorting key with a lambda. Here's an example:

```
python
# List of tuples
data = [(1, 5), (2, 3), (4, 1), (3, 2)]
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

# Sorting the list based on the second element of each tuple
sorted_data = sorted(data, key=lambda x: x[1])

print(sorted_data)

Output:

css
[(4, 1), (3, 2), (2, 3), (1, 5)]

Explanation:

- `key=lambda x: x[1]` specifies that the sorting should be based on the second element of each tuple (`x[1]`).
- `sorted()` returns a new sorted list, but you could also use the `sort()` method if you want to sort the list in place.

9. Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit.

You can use the `map()` function in Python to apply a conversion formula to each element of a list. To convert a temperature from Celsius to Fahrenheit, the formula is:

F=(C×95)+32F = \left(C \times \frac{9}{5} \right) + 32F=(C×59)+32

Here's how you can write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit:

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

```
python
# List of temperatures in Celsius
celsius_temperatures = [0, 20, 37, 100, -10]

# Function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

# Use map() to apply the conversion function to each element in
the list
fahrenheit_temperatures = list(map(celsius_to_fahrenheit,
celsius_temperatures))

# Print the converted list
print(fahrenheit_temperatures)
```

Output:

```
csharp
[32.0, 68.0, 98.6, 212.0, 14.0]
```

In this program:

- The `map()` function applies the `celsius_to_fahrenheit` function to each element of the `celsius_temperatures` list.
- The result is converted into a list and printed as the list of temperatures in Fahrenheit.

10. Create a Python program that uses `filter()` to remove all the vowels from a given string.

To create a Python program that uses `filter()` to remove all the vowels from a given string, you can follow these steps:

1. Define a function to check whether a character is a vowel.

PW SKILL
DATA FUNCTIONS ASSIGNENT (PRACTICAL)

2. Use `filter()` to filter out the vowels from the string.
3. Convert the result back to a string.

Here is a Python program that demonstrates this:

```
python
def remove_vowels(input_string):
    # Define a function that returns True for non-vowel
characters
    def is_not_vowel(char):
        return char.lower() not in 'aeiou'

    # Use filter to remove vowels and join the result into a new
string
    result = ''.join(filter(is_not_vowel, input_string))
    return result

# Example usage
input_string = "Hello, World!"
output_string = remove_vowels(input_string)
print(f"Original String: {input_string}")
print(f"String without vowels: {output_string}")
```

Explanation:

1. **`is_not_vowel` function**: This function checks whether a character is **not** a vowel by checking if the character (in lowercase) is not in the string `'aeiou'`.
2. **`filter()` function**: `filter()` applies the `is_not_vowel` function to each character of the input string. It returns an iterable with characters that are not vowels.
3. **`''.join()`**: This is used to combine the filtered characters back into a single string.

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

Example Output:

```
arduino
Original String: Hello, World!
String without vowels: Hll, Wrld!
```

This approach effectively removes all vowels from the string using `filter()`.

11. Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

34587, "Learning Python", "Mark Lutz", 4, 40.95

98762, "Programming Python", "Mark Lutz", 5, 56.80

77226, "Head First Python", "Paul Berry", 3, 32.95

88112, "Einführung in Python3", "Bernd Klein", 3, 24.99

The given structure is a list of sublists, where each sublist contains details about a book. To process such a dataset for accounting purposes, we can calculate total costs for each book entry, considering quantities and unit prices.

Here's what each sublist represents:

1. Order ID: A unique identifier for the order (e.g., 34587).
2. Book Title: The title of the book (e.g., "Learning Python").
3. Author Name: The author's name (e.g., "Mark Lutz").
4. Quantity: The number of copies ordered (e.g., 4).
5. Price per Unit: The cost of one unit of the book (e.g., 40.95).

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

Objective

The task might involve:

- Calculating the total cost for each order: Total Cost=Quantity×Price per Unit\text{Total Cost} = \text{Quantity} \times \text{Price per Unit}Total Cost=Quantity×Price per Unit
- Adding a minimum shipping cost if the total cost is below a certain threshold, e.g., $100.

Here's an example of a Python implementation for such a routine:

python

```
# Data

orders = [

    [34587, "Learning Python", "Mark Lutz", 4, 40.95],

    [98762, "Programming Python", "Mark Lutz", 5, 56.80],

    [77226, "Head First Python", "Paul Berry", 3, 32.95],

    [88112, "Einführung in Python3", "Bernd Klein", 3, 24.99]

]


# Processing routine

# Minimum total cost for free shipping is $100

MIN_SHIPPING_COST = 100
```

PW SKILL
DATA FUNCTIONS ASSIGNENT (PRACTICAL)

```
SHIPPING_FEE = 10


def calculate_total_cost(order):
    order_id, title, author, quantity, unit_price = order
    total_cost = quantity * unit_price
    # Add shipping fee if total cost is below the threshold
    if total_cost < MIN_SHIPPING_COST:
        total_cost += SHIPPING_FEE
    return (order_id, round(total_cost, 2))


# Calculate total costs for all orders
order_totals = list(map(calculate_total_cost, orders))


# Output
print(order_totals)
```

Expected Output

The program will return a list of tuples, each containing the Order ID and the total cost after considering shipping fees:

css

```
[    (34587, 173.8),    (98762, 284.0),    (77226, 109.85),    (88112, 84.97)]
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

(a) Write a Python program, which returns a list with 2-tuples. Each tuple consists of the order number and the product of the price per item and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €.

Here is the Python program:

python

def calculate_order_totals(orders):

    """

    Calculate the total price for each order

    Parameters:

    orders (list of tuples): Each tuple contains (order_number, price_per_item, quantity).

    Returns:

    list of tuples: Each tuple contains (order_number, total_price).

    """

    result = []

    for order in orders:

        order_number, price_per_item, quantity = order

```
        product = price_per_item * quantity
        if product < 100:
            product += 10  # Add €10 if the value is smaller than €100
        result.append((order_number, product))
    return result
# Example usage
orders = [
    (1, 20.0, 3),  # Order total = 20 * 3 = 60 < 100, so add €10
    (2, 50.0, 2),  # Order total = 50 * 2 = 100, no extra charge
    (3, 15.0, 5),  # Order total = 15 * 5 = 75 < 100, so add €10
]

totals = calculate_order_totals(orders)
print(totals)
```

Explanation:

1. Input: A list of tuples, where each tuple contains:
    - order_number: The identifier for the order.
    - price_per_item: The price for a single item.
    - quantity: The number of items in the order.

2. Logic:
   - Calculate the product of price_per_item and quantity for each order.
   - If the product is less than 100, add €10 to it.
3. Output: A list of tuples, where each tuple contains:
   - order_number: The same as in the input.
   - total_price: The adjusted total price for the order.

Example Output:

Plaintext

[(1, 70.0), (2, 100.0), (3, 85.0)]

(b) Write a Python program using lambda and map.

Here's an example Python program that demonstrates the use of lambda and map. This program takes a list of numbers and computes their squares:

python

```
# List of numbers

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



# Use map with a lambda function to compute the squares
```

PW SKILL
DATA FUNCTIONS ASSIGNENT
(PRACTICAL)

squared_numbers = list(map(lambda x: x**2, numbers))

Output the result

print("Original numbers:", numbers)

print("Squared numbers:", squared_numbers)

Explanation:

1. Input List: numbers contains the original list of integers.
2. Lambda Function: lambda x: x**2 defines an anonymous function that takes an input x and returns its square.
3. Map Function: map applies the lambda function to each element in numbers.
4. Convert to List: Since map returns a map object, we convert it to a list for better readability and usability.

Output:

less

Original numbers: [1, 2, 3, 4, 5]

Squared numbers: [1, 4, 9, 16, 25]