==

WEEK 1: VARIABLES AND DATA TYPES

==

Basic Data Types

Python Basic Data Types Overview

1. Integers (int):

- Whole numbers without decimal points
- No size limitation (can be as large as your memory allows)
- Examples: -5, 0, 1000

In [95]:
```
# Integer examples
age = 25
population = 1_000_000  # Underscores can be used for readability
print(f"Integers examples:\n- Age: {age}\n- Population: {population}")
```

```
Integers examples:
- Age: 25
- Population: 1000000
```

2. Floating-point numbers (float):

- Numbers with decimal points
- Also used for scientific notation
- Examples: 3.14, -0.001, 2.5e-4

In [96]:
```
# Float examples
pi = 3.14159
scientific_notation = 2.5e-4  # 0.00025
print(f"\nFloat examples:\n- Pi: {pi}\n- Scientific notation: {scientific_notation}")
```

```
Float examples:
- Pi: 3.14159
- Scientific notation: 0.00025
```

3. Strings (str):

- Sequence of characters
- Can use single or double quotes
- Can span multiple lines using triple quotes
- Immutable (cannot be changed after creation)

In [97]:
```
# String examples
single_quoted = 'Hello'
double_quoted = "World"
multi_line = """This is a
multi-line string"""
print(f"\nString examples:\n- Single quoted: {single_quoted}\n- Multi-line: {multi_line}")
```

```
String examples:
- Single quoted: Hello
- Multi-line: This is a
multi-line string
```

4. Booleans (bool):

- Can only have two values: True or False
- Used for logical operations

In [98]:
```python
# Boolean examples and operations
is_active = True
is_logged_in = False
# Boolean operations
is_valid_user = is_active and is_logged_in
print(f"\nBoolean examples:\n- Active: {is_active}\n- Valid user: {is_valid_user}")
```

```
Boolean examples:
- Active: True
- Valid user: False
```

Type conversion (Type Casting)

Common Type Conversions:

1. str() - Convert to string
2. int() - Convert to integer
3. float() - Convert to float
4. bool() - Convert to boolean

Note: Not all conversions are possible and may raise errors

In [99]:
```python
# String to number conversion
str_number = "123.45"
float_number = float(str_number)  # Convert string to float
int_number = int(float_number)    # Convert float to int
print(f"\nType conversion:\n- String to float: {float_number}\n- Float to int: {int_number}")
```

```
Type conversion:
- String to float: 123.45
- Float to int: 123
```

Variable naming conventions

Python Variable Naming Rules:

1. Must start with a letter or underscore
2. Can contain letters, numbers, and underscores
3. Case-sensitive
4. Cannot use Python keywords

Common Conventions:

1. snake_case for variables and functions
2. PascalCase for classes
3. UPPER_CASE for constants

In [100…
```python
user_name = "john_doe"      # Snake case (recommended for variables)
firstName = "John"          # Camel case (not recommended in Python)
MAX_ATTEMPTS = 3           # Upper case (used for constants)
```

String Operations

String Operations in Python:

1. String Formatting:

- f-strings (Python 3.6+)
- str.format() method
- % operator (older style)

In [101…
```python
# String formatting examples
name = "Alice"
age = 30
# f-string (recommended)
f_string = f"Name: {name}, Age: {age}"
# .format() method
format_string = "Name: {}, Age: {}".format(name, age)
# % operator (old style)
old_style = "Name: %s, Age: %d" % (name, age)
```

```
print("String Formatting Examples:")
print(f"f-string: {f_string}")
print(f"format(): {format_string}")
print(f"% operator: {old_style}")
```

```
String Formatting Examples:
f-string: Name: Alice, Age: 30
format(): Name: Alice, Age: 30
% operator: Name: Alice, Age: 30
```

2. String Methods:

- strip() - Remove whitespace
- split() - Split string into list
- join() - Join list into string
- upper()/lower() - Change case
- replace() - Replace substring

In [102…

```
# String methods
text = "  Hello, Python World!  "
print("\nString Methods:")
print(f"Original: '{text}'")
print(f"Stripped: '{text.strip()}'")
print(f"Upper: '{text.upper()}'")
print(f"Lower: '{text.lower()}'")
print(f"Replace: '{text.replace('Python', 'Amazing')}'")
print(f"Split: {text.split()}")
```

```
String Methods:
Original: '  Hello, Python World!  '
Stripped: 'Hello, Python World!'
Upper: '  HELLO, PYTHON WORLD!  '
Lower: '  hello, python world!  '
Replace: '  Hello, Amazing World!  '
Split: ['Hello,', 'Python', 'World!']
```

3. String Slicing:

- Basic slicing: string[start:end:step]
- Negative indices count from end
- Omitting indices uses defaults

In [103…

```
# String slicing
text = "Python Programming"
print("\nString Slicing:")
print(f"Original: {text}")
print(f"First 6 chars: {text[0:6]}")        # Python
print(f"Last 11 chars: {text[-11:]}")       # Programming
print(f"Every 2nd char: {text[::2]}")       # Pto rgamn
print(f"Reverse: {text[::-1]}")             # gnimmargorP nohtyP
```

```
String Slicing:
Original: Python Programming
First 6 chars: Python
Last 11 chars: Programming
Every 2nd char: Pto rgamn
Reverse: gnimmargorP nohtyP
```

Numeric Operations and Types of Operators

Numeric Operations in Python:

*1. Arithmetic Operators:**

- (+) : Addition
- (-) : Subtraction
- (*) : Multiplication
- (/) : Division (returns float)
- (//) : Floor division (returns int)
- (%) : Modulus (remainder)
- (**) : Exponentiation

In [104…

```
# Basic arithmetic
```

```python
a, b = 15, 4
print("Arithmetic Operators:")
print(f"a = {a}, b = {b}")
print(f"Addition: {a} + {b} = {a + b}")
print(f"Subtraction: {a} - {b} = {a - b}")
print(f"Multiplication: {a} * {b} = {a * b}")
print(f"Division: {a} / {b} = {a / b}")
print(f"Floor Division: {a} // {b} = {a // b}")
print(f"Modulus: {a} % {b} = {a % b}")
print(f"Exponentiation: {a} ** {b} = {a ** b}")
```

```
Arithmetic Operators:
a = 15, b = 4
Addition: 15 + 4 = 19
Subtraction: 15 - 4 = 11
Multiplication: 15 * 4 = 60
Division: 15 / 4 = 3.75
Floor Division: 15 // 4 = 3
Modulus: 15 % 4 = 3
Exponentiation: 15 ** 4 = 50625
```

2. Assignment Operators:

Simple Assignment (=)

- Used to assign values to variables
- Assigns the value on the right to the variable on the left
    - Example: x = 5 assigns the value 5 to variable x

Addition Assignment (+=)

- Adds the right operand to the left operand and assigns the result to the left operand
- Equivalent to: x = x + y
- Commonly used in loops and counters
    - Example: If x = 5, then x += 3 results in x = 8

Subtraction Assignment (-=)

- Subtracts the right operand from the left operand and assigns the result to the left operand
- Equivalent to: x = x - y
- Useful for decremental operations
    - Example: If x = 8, then x -= 3 results in x = 5

Multiplication Assignment (*=)

- Multiplies the left operand by the right operand and assigns the result to the left operand
- Equivalent to: x = x * y
- Common in scaling operations
    - Example: If x = 4, then x *= 2 results in x = 8

Division Assignment (/=)

- Divides the left operand by the right operand and assigns the result to the left operand
- Equivalent to: x = x / y
- Always performs float division
    - Example: If x = 8, then x /= 2 results in x = 4.0

Floor Division Assignment (//=)

- Performs floor division and assigns the result to the left operand
- Equivalent to: x = x // y
- Returns the largest integer less than or equal to the division result
    - Example: If x = 8, then x //= 3 results in x = 2

Modulus Assignment (%=)

- Performs modulus operation and assigns the remainder to the left operand

- Equivalent to: x = x % y
- Used for getting remainders or cycling through values
  - Example: If x = 8, then x %= 3 results in x = 2

Exponentiation Assignment (=)**

- Raises the left operand to the power of the right operand and assigns the result
- Equivalent to: x = x ** y
- Used for exponential calculations
  - Example: If x = 2, then x **= 3 results in x = 8

In [105…

```
# Assignment operators
x = 10
print("\nAssignment Operators:")
print(f"Initial x: {x}")
x += 5  # x = x + 5
print(f"x += 5: {x}")
x -= 3  # x = x - 3
print(f"x -= 3: {x}")
x *= 2  # x = x * 2
print(f"x *= 2: {x}")
x /= 4  # x = x / 4
print(f"x /= 4: {x}")
x **= 2  # x = x ** 2
print(f"x **= 2: {x}")
```

```
Assignment Operators:
Initial x: 10
x += 5: 15
x -= 3: 12
x *= 2: 24
x /= 4: 6.0
x **= 2: 36.0
```

3. Comparison Operators:

== : Equal to

!= : Not equal to

** > : Greater than**

< : Less than

>= : Greater than or equal to

<= : Less than or equal to

In [106…

```
# Comparison operators
print("\nComparison Operators:")
print(f"{a} == {b}: {a == b}")
print(f"{a} != {b}: {a != b}")
print(f"{a} > {b}: {a > b}")
print(f"{a} < {b}: {a < b}")
print(f"{a} >= {b}: {a >= b}")
print(f"{a} <= {b}: {a <= b}")
```

```
Comparison Operators:
15 == 4: False
15 != 4: True
15 > 4: True
15 < 4: False
15 >= 4: True
15 <= 4: False
```

4. Logical Operators:

and

- True if both statements are true

or

- True if at least one statement is true

not

- Inverts the boolean value

In [107…

```
# Logical operators
print("\nLogical Operators:")
print(f"({a} > 10) and ({b} < 5): {(a > 10) and (b < 5)}")
print(f"({a} > 20) or ({b} < 5): {(a > 20) or (b < 5)}")
print(f"not ({a} == 15): {not (a == 15)}")
```

```
Logical Operators:
(15 > 10) and (4 < 5): True
(15 > 20) or (4 < 5): True
not (15 == 15): False
```

5. Bitwise Operators:

& (AND):

- The AND operator performs a bitwise comparison between each bit of two integers. If both corresponding bits are 1, the result is 1; otherwise, it's 0. This is useful for masking specific bits in a binary number.
- Example: 5 & 3 results in 1 because only the least significant bit is 1 in both numbers.

| (OR):

- The OR operator compares each bit of two integers. If at least one of the corresponding bits is 1, the result is 1; otherwise, it's 0. This operator can be used to set specific bits in a binary number.
- Example: 5 | 3 results in 7, as all bits that are 1 in either number are set to 1.

^ (XOR):

- The XOR (exclusive OR) operator outputs 1 if the bits differ (one is 1, the other is 0); otherwise, it outputs 0. This is often used for bit manipulation tasks like toggling bits.
- Example: 5 ^ 3 results in 6, as the differing bits are set to 1.

~ (NOT):

- The NOT operator is a bitwise negation operator that inverts all bits in a binary number: 0 becomes 1 and 1 becomes 0. It effectively returns the two's complement of the number, changing the sign.
- Example: ~5 results in -6 in most systems due to the way signed numbers are represented in binary (two's complement notation).

<< (Left Shift):

- The Left Shift operator shifts all bits in a binary number to the left by a specified number of positions, filling the vacant positions with 0s on the right. Each left shift by 1 position effectively multiplies the number by 2.
- Example: 5 << 1 results in 10, as all bits are shifted one position to the left.

>> (Right Shift):

- The Right Shift operator shifts all bits in a binary number to the right by a specified number of positions. For unsigned numbers, the vacant positions are filled with 0s on the left. For signed numbers, the behavior may vary, depending on the system.
- Example: 5 >> 1 results in 2, as all bits are shifted one position to the right.

In [108…

```
# Bitwise operators
print("\nBitwise Operators:")
```

```python
print(f"{a} & {b}: {a & b}")
print(f"{a} | {b}: {a | b}")
print(f"{a} ^ {b}: {a ^ b}")
print(f"~{a}: {~a}")
print(f"{a} << 1: {a << 1}")
print(f"{a} >> 1: {a >> 1}")
```

```
Bitwise Operators:
15 & 4: 4
15 | 4: 15
15 ^ 4: 11
~15: -16
15 << 1: 30
15 >> 1: 7
```

6. Identity Operators:

is :

- True if both variables are the same object

is not :

- True if both variables are not the same object

In [109…

```python
# Identity operators
print("\nIdentity Operators:")
print(f"{a} is {b}: {a is b}")
print(f"{a} is not {b}: {a is not b}")
```

```
Identity Operators:
15 is 4: False
15 is not 4: True
```

7. Membership Operators:

in :

- True if a sequence contains the specified element

not in :

- True if a sequence does not contain the specified element

In [110…

```python
# Membership operators
numbers = [1, 2, 3, 4, 5]
print("\nMembership Operators:")
print(f"3 in numbers: {3 in numbers}")
print(f"6 not in numbers: {6 not in numbers}")
```

```
Membership Operators:
3 in numbers: True
6 not in numbers: True
```

Lists

1. List Properties:

- Ordered collection of items
- Mutable (can be modified)
- Can contain mixed data types
- Created using square brackets []

In [111…

```python
# List creation and basic operations
fruits = ["apple", "banana", "orange"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]  # Mixed data types

print("List Operations:")
print(f"Original fruits: {fruits}")
print(f"Original numbers: {numbers}")
```

```
print(f"Mixed list: {mixed}")
```

```
List Operations:
Original fruits: ['apple', 'banana', 'orange']
Original numbers: [1, 2, 3, 4, 5]
Mixed list: [1, 'hello', 3.14, True]
```

2. Common List Methods:

- append() - Add item to end
- extend() - Add items from iterable
- insert() - Add item at position
- remove() - Remove specific item
- pop() - Remove and return item
- sort() - Sort list in place
- reverse() - Reverse list in place

In [112…

```
# List methods
fruits.append("grape")              # Add single item
fruits.extend(["kiwi", "mango"])   # Add multiple items
fruits.insert(1, "pear")           # Insert at position

print("\nList Methods:")
print(f"After append and extend: {fruits}")
removed_fruit = fruits.pop()       # Remove and return last item
print(f"Removed fruit: {removed_fruit}")
print(f"After pop: {fruits}")
```

```
List Methods:
After append and extend: ['apple', 'pear', 'banana', 'orange', 'grape', 'kiwi', 'mango']
Removed fruit: mango
After pop: ['apple', 'pear', 'banana', 'orange', 'grape', 'kiwi']
```

3. List Operations:

- Indexing (positive and negative)
- Slicing [start:end:step]
- Concatenation (+)
- Multiplication (*)

In [113…

```
# List slicing and indexing
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("\nList Slicing:")
print(f"Original: {numbers}")
print(f"First 3: {numbers[:3]}")
print(f"Last 3: {numbers[-3:]}")
print(f"Every 2nd number: {numbers[::2]}")
print(f"Reversed: {numbers[::-1]}")
```

```
List Slicing:
Original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
First 3: [0, 1, 2]
Last 3: [7, 8, 9]
Every 2nd number: [0, 2, 4, 6, 8]
Reversed: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

List comprehension

List Comprehension Syntax:

- [expression for item in iterable if condition]
- **expression:** The value or operation you want to include in the resulting list. It could be a transformation or computation on the item.
- **item:** The variable representing each element in the iterable as the loop progresses.
- **iterable:** A collection (like a list, tuple, or string) that you're iterating over.
- **if condition:** An optional filtering condition that ensures only elements satisfying the condition are included.
- **Result:** A new list is generated by evaluating the expression for each item in the iterable that meets the condition.

In [114…
```
squares = [x**2 for x in range(10) if x % 2 == 0]
print("\nList Comprehension:")
print(f"Squares of even numbers: {squares}")

# Nested lists (2D lists)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("\nNested Lists:")
print("Matrix:")
for row in matrix:
    print(row)
```

```
List Comprehension:
Squares of even numbers: [0, 4, 16, 36, 64]

Nested Lists:
Matrix:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

Tuples

1. Tuple Properties:

- Ordered collection of items
- Immutable (cannot be modified)
- Created using parentheses ()
- Can contain mixed data types

In [115…
```
# Basic tuple operations
coordinates = (10, 20)
rgb_color = (255, 128, 0)
mixed_tuple = (1, "hello", 3.14)

print("Tuple Operations:")
print(f"Coordinates: {coordinates}")
print(f"RGB Color: {rgb_color}")
print(f"Mixed tuple: {mixed_tuple}")
```

```
Tuple Operations:
Coordinates: (10, 20)
RGB Color: (255, 128, 0)
Mixed tuple: (1, 'hello', 3.14)
```

2. Tuple vs Lists:

- Tuples are immutable
- Tuples are faster than lists
- Tuples can be dictionary keys
- Tuples are good for fixed data

In [116…
```
# Tuple unpacking
x, y = coordinates
r, g, b = rgb_color
print("\nTuple Unpacking:")
print(f"x: {x}, y: {y}")
print(f"Red: {r}, Green: {g}, Blue: {b}")
```

```
Tuple Unpacking:
x: 10, y: 20
Red: 255, Green: 128, Blue: 0
```

3. Named Tuples:

- From collections module
- Adds meaning to tuple positions
- Creates class-like objects

In [117…
```
# Named tuples
from collections import namedtuple
```

```python
Point = namedtuple('Point', ['x', 'y', 'z'])
point = Point(1, 2, 3)
print("\nNamed Tuples:")
print(f"Point: {point}")
print(f"X coordinate: {point.x}")
print(f"As list: {list(point)}")
```

```
Named Tuples:
Point: Point(x=1, y=2, z=3)
X coordinate: 1
As list: [1, 2, 3]
```

Dictionaries

1. Dictionary Properties:

- Unordered collection (Python 3.7+ maintains insertion order)
- Key-value pairs
- Mutable
- Keys must be immutable (strings, numbers, tuples)

In [118…

```python
# Dictionary creation and basic operations
person = {
    "name": "John Doe",
    "age": 30,
    "city": "New York",
    "languages": ["Python", "JavaScript"]
}

print("Dictionary Operations:")
print(f"Person dictionary: {person}")

# Accessing and modifying
print("\nAccessing Dictionary:")
print(f"Name: {person['name']}")
print(f"Age: {person.get('age')}")  # Safe access
print(f"Country: {person.get('country', 'Not specified')}")  # With default
```

```
Dictionary Operations:
Person dictionary: {'name': 'John Doe', 'age': 30, 'city': 'New York', 'languages': ['Python', 'JavaScript']}

Accessing Dictionary:
Name: John Doe
Age: 30
Country: Not specified
```

2. Common Dictionary Methods:

- keys() - Return keys view
- values() - Return values view
- items() - Return key-value pairs
- get() - Safe way to get values
- update() - Update dictionary
- pop() - Remove and return value

In [119…

```python
# Dictionary methods
print("\nDictionary Methods:")
print(f"Keys: {list(person.keys())}")
print(f"Values: {list(person.values())}")
print(f"Items: {list(person.items())}")
```

```
Dictionary Methods:
Keys: ['name', 'age', 'city', 'languages']
Values: ['John Doe', 30, 'New York', ['Python', 'JavaScript']]
Items: [('name', 'John Doe'), ('age', 30), ('city', 'New York'), ('languages', ['Python', 'JavaScript'])]
```

3. Dictionary Comprehension:

{key_expr: value_expr for item in iterable}

- Creates a dictionary by applying key_expr and value_expr to each item in iterable.
- Efficient way to construct dictionaries in a single line using comprehension syntax.

In [120…

```python
# Dictionary comprehension
```

```python
squares_dict = {x: x**2 for x in range(5)}
print("\nDictionary Comprehension:")
print(f"Squares dictionary: {squares_dict}")

# Nested dictionaries
companies = {
    "tech": {
        "name": "TechCorp",
        "employees": 1000,
        "locations": ["NY", "SF"]
    },
    "retail": {
        "name": "RetailCo",
        "employees": 500,
        "locations": ["LA", "Chicago"]
    }
}

print("\nNested Dictionaries:")
print(f"Tech company employees: {companies['tech']['employees']}")
```

```
Dictionary Comprehension:
Squares dictionary: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Nested Dictionaries:
Tech company employees: 1000
```

Sets

1. Set Properties:

- Unordered collection
- Unique elements only
- Mutable
- Elements must be immutable
- Created using {} or set()

In [121…

```python
# Set creation and basic operations
fruits = {"apple", "banana", "orange"}
vegetables = {"carrot", "lettuce", "cucumber"}
numbers = {1, 2, 3, 4, 5}

print("Set Operations:")
print(f"Fruits set: {fruits}")
print(f"Numbers set: {numbers}")
```

```
Set Operations:
Fruits set: {'banana', 'apple', 'orange'}
Numbers set: {1, 2, 3, 4, 5}
```

2. Set Operations:

- union (|) - Combine sets
- intersection (&) - Common elements
- difference (-) - Elements in first but not second
- symmetric_difference (^) - Elements in either but not both

In [122…

```python
# Adding and removing elements
fruits.add("grape")
fruits.discard("banana")
print("\nModified Sets:")
print(f"After add/discard: {fruits}")
```

```
Modified Sets:
After add/discard: {'grape', 'apple', 'orange'}
```

3. Common Set Methods:

- add() - Add element
- remove() - Remove element (raises error if missing)
- discard() - Remove element (no error if missing)
- update() - Add elements from iterable

In [123…

```python
# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

print("\nSet Operations:")
print(f"Set1: {set1}")
print(f"Set2: {set2}")
print(f"Union: {set1 | set2}")
print(f"Intersection: {set1 & set2}")
print(f"Difference: {set1 - set2}")
print(f"Symmetric Difference: {set1 ^ set2}")
```

```
Set Operations:
Set1: {1, 2, 3, 4}
Set2: {3, 4, 5, 6}
Union: {1, 2, 3, 4, 5, 6}
Intersection: {3, 4}
Difference: {1, 2}
Symmetric Difference: {1, 2, 5, 6}
```

Arrays and NumPy

1. Python Arrays:

- From array module
- Homogeneous data
- More memory efficient than lists
- Limited functionality

2. NumPy Arrays:

- More powerful than Python arrays
- Multi-dimensional
- Vectorized operations
- Many mathematical functions
- Essential for data science

In [124…

```python
import numpy as np

# Basic NumPy array operations
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([10, 20, 30, 40, 50])

print("NumPy Array Operations:")
print(f"Array 1: {arr1}")
print(f"Array 2: {arr2}")
print(f"Sum: {arr1 + arr2}")
print(f"Product: {arr1 * arr2}")
print(f"Difference: {arr2 - arr1}")
print(f"Division: {arr2 / arr1}")
print(f"Dot Product: {np.dot(arr1, arr2)}")
print(f"Mean of Array 1: {np.mean(arr1)}")
print(f"Standard Deviation of Array 2: {np.std(arr2)}")
```

```
NumPy Array Operations:
Array 1: [1 2 3 4 5]
Array 2: [10 20 30 40 50]
Sum: [11 22 33 44 55]
Product: [ 10  40  90 160 250]
Difference: [ 9 18 27 36 45]
Division: [10. 10. 10. 10. 10.]
Dot Product: 550
Mean of Array 1: 3.0
Standard Deviation of Array 2: 14.142135623730951
```

Control Structures

1. Conditional Statements:

- **if:** Basic conditional
- **elif:** Else if condition
- **else:** Default case

- Conditional expressions (ternary operator)

In [125…
```
# Basic conditional statements
age = 18
income = 50000
```

2. Comparison Operators:

- **==:** Equal to
- **!=:** Not equal to
- **>, <:** Greater than, Less than
- **>=, <=:** Greater than or equal to, Less than or equal to
- **is:** Identity comparison
- **in:** Membership test

In [126…
```
print("Conditional Statements:")
if age >= 18:
    if income >= 40000:
        print("Eligible for premium card")
    else:
        print("Eligible for basic card")
else:
    print("Not eligible for card")
```

```
Conditional Statements:
Eligible for premium card
```

3. Logical Operators:

- **and:** Both conditions true
- **or:** At least one condition true
- **not:** Negation

In [127…
```
# Conditional expression (ternary operator)
status = "adult" if age >= 18 else "minor"
print(f"\nStatus using ternary: {status}")
```

```
Status using ternary: adult
```

Loops in Python:

1. For Loops:

- **Iterate over sequences:** Looping through elements in a list, tuple, or string.
- **range() function:** Generates a sequence of numbers, often used in loops.
- **enumerate() function:** Adds a counter to an iterable and returns index-value pairs.
- **zip() function:** Combines multiple iterables element-wise into tuples.

In [128…
```
# For loop examples
print("\nFor Loop Examples:")
# Iterating over a list
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(f"Processing {fruit}")

# Using range
for i in range(3):
    print(f"Iteration {i}")

# Using enumerate
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

```
For Loop Examples:
Processing apple
Processing banana
Processing orange
Iteration 0
Iteration 1
Iteration 2
Index 0: apple
Index 1: banana
Index 2: orange
```

2. While Loops:

- **Condition-based iteration:** Looping through data only when certain conditions are met.
- **break statement:** Exits a loop prematurely when a condition is satisfied.
- **continue statement:** Skips the current loop iteration and moves to the next.
- **else clause:** Executes code after a loop finishes, if no break occurs.

In [129…

```
# While loop examples
print("\nWhile Loop Examples:")
count = 0
while count < 3:
    print(f"Count: {count}")
    count += 1
```

```
While Loop Examples:
Count: 0
Count: 1
Count: 2
```

3. Loop Control:

- **break:** Exit loop
- **continue:** Skip to next iteration
- **pass:** Do nothing placeholder

In [130…

```
# Break and continue
print("\nBreak and Continue Examples:")
for i in range(5):
    if i == 2:
        continue  # Skip 2
    if i == 4:
        break     # Stop at 4
    print(f"Value: {i}")
```

```
Break and Continue Examples:
Value: 0
Value: 1
Value: 3
```

Functions

1. Function Definition:

- **def keyword:** Used to define a new function in Python.
- **Parameters vs Arguments:** Parameters are placeholders; arguments are values passed to functions.
- **Default parameters:** Parameters with predefined values if arguments are not given.
- **Return values:** The output values that a function sends back.
- **Docstrings:** Descriptive text for functions or modules, aiding documentation.

In [131…

```
# Basic function definition
def calculate_total(items, tax_rate=0.1):
    """
    Calculate total cost including tax.

    Args:
        items (list): List of item prices
        tax_rate (float): Tax rate as decimal (default 0.1)

    Returns:
        float: Total cost including tax
    """
    subtotal = sum(items)
    tax = subtotal * tax_rate
    return subtotal + tax

# Function usage
prices = [10, 20, 30]
total = calculate_total(prices)
print(f"Function Example:")
print(f"Total with default tax: ${total:.2f}")
print(f"Total with 20% tax: ${calculate_total(prices, 0.2):.2f}")
```

```
Function Example:
Total with default tax: $66.00
Total with 20% tax: $72.00
```

2. Parameter Types:

- **Positional arguments**: Arguments passed in order, matching function parameter positions.
- **Keyword arguments**: Arguments passed by name, allowing out-of-order placement.
- **Default parameters**: Parameters with preset values if no argument is provided.
- **Variable-length arguments (*args)**: Collects multiple positional arguments into a tuple.
- **Keyword variable-length arguments (kwargs)**: Collects multiple keyword arguments into a dictionary.

In [132…

```
# Args and kwargs
def flexible_function(*args, **kwargs):
    """
    Demonstrate variable arguments and keyword arguments.
    """
    print("\nFlexible Function Example:")
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

flexible_function(1, 2, 3, name="John", age=30)
```

```
Flexible Function Example:
Positional arguments: (1, 2, 3)
Keyword arguments: {'name': 'John', 'age': 30}
```

3. Function Features:

- **Lambda functions:** Anonymous, single-line functions defined with the lambda keyword.
- **Recursion:** A function calling itself to solve a smaller problem.
- **Decorators:** Functions modifying another function's behavior without altering it.
- **Generator functions:** Functions yielding values one at a time using yield.

In [133…

```
# Lambda function
square = lambda x: x**2
print("\nLambda Function Example:")
print(f"Square of 5: {square(5)}")

# Generator function
def number_generator(n):
    """Generate numbers up to n."""
    for i in range(n):
        yield i

print("\nGenerator Function Example:")
gen = number_generator(3)
for num in gen:
    print(f"Generated: {num}")
```

```
Lambda Function Example:
Square of 5: 25

Generator Function Example:
Generated: 0
Generated: 1
Generated: 2
```

Object-Oriented Programming

1. Classes and Objects:

- **Class definition:** Blueprint for creating objects, defined using the class keyword.
- **Instance creation:** Process of creating an object from a class.
- **Instance methods:** Functions acting on an instance, accessed via self parameter.
- **Class methods:** Functions acting on class itself, accessed via cls.
- **Static methods:** Independent functions within a class, unrelated to instance or class.
- **Properties:** Attributes accessed like variables but implemented with getter/setter methods.

In [134…

```
# Basic class definition
class BankAccount:
    """A simple bank account class."""

    interest_rate = 0.02  # Class variable

    def __init__(self, owner, balance=0):
        self._owner = owner      # Protected attribute
```

```python
        self.__balance = balance # Private attribute

    @property
    def balance(self):
        """Get account balance."""
        return self.__balance

    def deposit(self, amount):
        """Deposit money."""
        if amount > 0:
            self.__balance += amount
            return True
        return False

    @classmethod
    def set_interest_rate(cls, rate):
        """Set interest rate for all accounts."""
        cls.interest_rate = rate

    @staticmethod
    def validate_amount(amount):
        """Validate transaction amount."""
        return amount > 0
```

2. Inheritance:

- **Single inheritance:** A class inherits from only one parent class.
- **Multiple inheritance:** A class inherits from more than one parent class.
- **Method overriding:** Redefining a parent class method in the subclass.
- **super() function:** Calls a parent class method or constructor in subclass.

In [135…

```python
# Inheritance example
class SavingsAccount(BankAccount):
    """A savings account with withdrawal limits."""

    def __init__(self, owner, balance=0, withdrawal_limit=1000):
        super().__init__(owner, balance)
        self.withdrawal_limit = withdrawal_limit

    def withdraw(self, amount):
        """Withdraw money with limit check."""
        if amount <= self.withdrawal_limit:
            if self.balance >= amount:
                self._BankAccount__balance -= amount  # Accessing private attribute
                return True
        return False
```

3. Encapsulation:

- **Private attributes (_):** Attributes with limited access, conventionally using a single underscore.
- **Name mangling (__):** Alters attribute names to prevent accidental access in subclasses.
- **Property decorators:** Simplify getter/setter methods using @property and @name.setter.
- **Getter/setter methods:** Control attribute access and modification with encapsulation.

In [136…

```python
# Using the classes
print("OOP Examples:")
account = SavingsAccount("John Doe", 1000)
account.deposit(500)
print(f"Balance after deposit: ${account.balance}")
account.withdraw(200)
print(f"Balance after withdrawal: ${account.balance}")
```

```
OOP Examples:
Balance after deposit: $1500
Balance after withdrawal: $1300
```

Error Handling

Error Handling in Python:

1. Try-Except Structure:

- try block: Potential error code

- except block: Handle specific errors
- else block: Execute if no error
- finally block: Always execute

In [137…

```
# Basic error handling
def divide_numbers(a, b):
    """Divide two numbers with error handling."""
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Division by zero!")
        return None
    except TypeError:
        print("Error: Invalid number type!")
        return None
    else:
        print("Division successful!")
        return result
    finally:
        print("Division operation completed.")
```

2. Error Handling

- **ValueError:** Raised when a function receives an inappropriate value.
- **TypeError:** Raised when an operation is applied to incompatible types.
- **IndexError:** Raised when accessing an index outside a sequence's range.
- **KeyError:** Raised when a nonexistent dictionary key is accessed.
- **ZeroDivisionError:** Raised when attempting to divide by zero.
- **FileNotFoundError:** Raised when attempting to access a nonexistent file.

In [138…

```
# Testing error handling
print("\nError Handling Examples:")
print(f"10/2 = {divide_numbers(10, 2)}")
print(f"10/0 = {divide_numbers(10, 0)}")
print(f"10/'2' = {divide_numbers(10, '2')}")
```

```
Error Handling Examples:
Division successful!
Division operation completed.
10/2 = 5.0
Error: Division by zero!
Division operation completed.
10/0 = None
Error: Invalid number type!
Division operation completed.
10/'2' = None
```

3. Custom Exceptions:

- **Creating custom exceptions:** Define unique exceptions by subclassing Python's Exception class.
- **Raising exceptions:** Use raise to trigger exceptions when specific conditions occur.
- **Exception hierarchy:** Organized structure of built-in exceptions from base Exception class.

In [139…

```
# Custom exception
class InsufficientFundsError(Exception):
    """Custom exception for insufficient funds."""
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        self.message = f"Insufficient funds: balance=${balance}, required=${amount}"
        super().__init__(self.message)

# Using custom exception
def withdraw(balance, amount):
    """Withdraw with custom exception."""
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
    return balance - amount

# Testing custom exception
try:
    result = withdraw(100, 200)
except InsufficientFundsError as e:
    print(f"\nCustom Exception Example:\n{e}")
```

```
Custom Exception Example:
Insufficient funds: balance=$100, required=$200
```

Additional Important Python Topics

1. File Handling:

- Reading files
- Writing files
- Context managers (with statement)
- JSON handling

In [140…

```
# File handling example
print("\nFile Handling Example:")
# Writing to file
with open("example.txt", "w") as f:
    f.write("Hello, Python!")

# Reading from file
with open("example.txt", "r") as f:
    content = f.read()
    print(f"File content: {content}")

# JSON handling
import json

data = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# Convert to JSON string
json_string = json.dumps(data, indent=2)
print("\nJSON Example:")
print(json_string)
```

```
File Handling Example:
File content: Hello, Python!

JSON Example:
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
```

2. Regular Expressions:

- **Pattern matching:** Searching for specific sequences in data using predefined rules.
- **re module:** Python library for working with regular expressions and pattern matching.
- **Common patterns:** Frequent regular expressions used for email, phone numbers, dates, etc.

3. Modules and Packages:

- **Creating modules:**Writing Python code in separate files for reusability.
- **Importing modules:** Bringing external Python code into your program for use.
- **Package structure:** Organizing Python files into directories with an **init**.py file.

In [141…

```
# Regular expression example
import re

text = "Python programming: python@example.com"
email_pattern = r'\b[\w\.-]+@[\w\.-]+\.\w+\b'

print("\nRegex Example:")
if re.search(email_pattern, text):
    email = re.findall(email_pattern, text)[0]
    print(f"Found email: {email}")
```

```
Regex Example:
Found email: python@example.com
```