Learn Python Programming from Scratch

Topic: Comments in Python

1. What are Comments?

Comments are lines of text in your code that are completely ignored by the Python interpreter. They exist solely for human readers to understand the code better.

Comments serve several important purposes:

- **Explain complex logic**: Help others (and your future self) understand what the code does
- **Document functions and classes**: Provide usage instructions and parameter descriptions
- **Make notes**: Leave reminders or important information about the code
- **Temporarily disable code**: Comment out code during debugging or testing
- **Provide context**: Explain why certain decisions were made in the code

2. Why Comments are Essential

Good comments are crucial for:

- **Code maintenance**: Making it easier to update and fix code later
- **Team collaboration**: Helping other developers understand your code
- **Learning**: Documenting your thought process as you learn
- **Debugging**: Temporarily disabling problematic code
- **Professional development**: Writing clean, well-documented code

3. Types of Comments in Python

Python supports several ways to add comments to your code:

Single-line Comments (`#`)

- Start with the hash/pound symbol `#`
- Everything after `#` on that line is ignored by Python
- Used for brief explanations or notes

Multi-line Comments (`"""` or `'''`)

- Use triple quotes (double or single)
- Can span multiple lines
- Often used for longer explanations or documentation

Inline Comments

- Comments placed at the end of a line of code

- Should be used sparingly and with proper spacing
- Need at least two spaces between code and `#`

Docstrings

- Special multi-line strings used to document modules, functions, and classes
- Accessible at runtime via the `__doc__` attribute

In [1]:
```
# Single-line Comments Examples

print("=== SINGLE-LINE COMMENTS ===")

# This is a single-line comment
# Comments are completely ignored by the Python interpreter
# You can write anything here - it won't affect your program

print("This line will execute normally")

# You can have multiple single-line comments in a row
# Each line needs its own # symbol at the beginning
# This is useful for longer explanations that span multiple lines

# Comments can also be used to add context to your variables
user_name = "Alice"      # Store the current user's name
user_age = 25            # Store the user's age in years
is_premium = True        # Track if user has premium membership

print(f"User: {user_name}, Age: {user_age}, Premium: {is_premium}")

# Comments are great for explaining complex calculations
pi = 3.14159             # Mathematical constant pi
radius = 5               # Circle radius in meters
area = pi * radius ** 2  # Calculate area using formula: π × r²

print(f"Circle area with radius {radius}m: {area:.2f} square meters")

print("\n" + "="*60)

# Inline comments (use sparingly!)
x = 5    # This is an inline comment
y = 10   # Another inline comment
result = x * y  # Calculate product

print(f"Result: {result}")

# You can temporarily disable code by commenting it out
print("This line runs")
# print("This line is commented out and won't run")
print("This line also runs")

print("\n=== COMMENTING OUT CODE FOR DEBUGGING ===")
# This is useful when testing or debugging
total = 100
# discount = 20  # Temporarily disable discount
discount = 0     # Use this instead for testing
final_price = total - discount
print(f"Final price: ${final_price}")
```

```
=== SINGLE-LINE COMMENTS ===
This line will execute normally
User: Alice, Age: 25, Premium: True
Circle area with radius 5m: 78.54 square meters

============================================================
Result: 50
This line runs
This line also runs

=== COMMENTING OUT CODE FOR DEBUGGING ===
Final price: $100
```

4. Multi-line Comments and Docstrings

In [7]:

```python
# Multi-line Comments and Docstrings

print("=== MULTI-LINE COMMENTS ===")

"""
This is a multi-line comment using triple double quotes.
It can span multiple lines and is very useful for:
- Longer explanations
- Commenting out blocks of code
- Adding detailed documentation
- Writing instructions or notes

Everything between the triple quotes is ignored by Python.
"""

'''
This is another multi-line comment using triple single quotes.
Both triple double quotes """ and triple single quotes ('' ')
work exactly the same way in Python.
You can choose either style based on your preference.
'''

print("Multi-line comments demonstrated above")

print("\n=== DOCSTRINGS FOR FUNCTIONS ===")

def calculate_circle_area(radius):
    """
    Calculate the area of a circle given its radius.

    This function uses the mathematical formula: Area = π × r²
    where π (pi) is approximately 3.14159 and r is the radius.

    Parameters:
    -----------
    radius : float or int
        The radius of the circle in any unit of measurement

    Returns:
    --------
    float
        The area of the circle in square units

    Examples:
```

```
    ---------
    >>> calculate_circle_area(5)
    78.53975

    >>> calculate_circle_area(10)
    314.159

    Note:
    -----
    This function assumes the input radius is positive.
    Negative values will still calculate but may not represent real areas.
    """
    pi = 3.14159
    area = pi * radius ** 2
    return area

# Test the function
area1 = calculate_circle_area(5)
area2 = calculate_circle_area(10)

print(f"Circle with radius 5: {area1:.2f} square units")
print(f"Circle with radius 10: {area2:.2f} square units")

# Access the docstring
print(f"\nFunction documentation:")
print(calculate_circle_area.__doc__)

print("\n=== COMMENTING OUT CODE BLOCKS ===")

"""
You can use multi-line comments to temporarily disable
entire blocks of code. This is useful for:

# Debugging - disable problematic sections
# Testing - try different approaches
# Development - keep alternative implementations

Here's an example of commented-out code:

def old_calculation(x, y):
    # This was our old method
    return x + y * 2

def alternative_approach():
    # This is an alternative we might try later
    pass
"""

print("The code block above is completely ignored")

print("\n=== MODULE DOCSTRING EXAMPLE ===")

# This would typically be at the top of a Python file
"""
shopping_cart.py - Shopping Cart Management System

This module provides functionality for managing a shopping cart,
including adding items, calculating totals, and applying discounts.

Classes:
```

```
--------
ShoppingCart: Main class for cart operations

Functions:
----------
calculate_tax(amount, rate): Calculate tax on a given amount
apply_discount(total, discount_percent): Apply percentage discount

Author: Prakash Ukhalkar
Version: 1.0
Date: 2024
"""

print("Module-level documentation demonstrated above")
```

```
=== MULTI-LINE COMMENTS ===
Multi-line comments demonstrated above

=== DOCSTRINGS FOR FUNCTIONS ===
Circle with radius 5: 78.54 square units
Circle with radius 10: 314.16 square units

Function documentation:

Calculate the area of a circle given its radius.

This function uses the mathematical formula: Area = π × r²
where π (pi) is approximately 3.14159 and r is the radius.

Parameters:
-----------
radius : float or int
    The radius of the circle in any unit of measurement

Returns:
--------
float
    The area of the circle in square units

Examples:
---------
>>> calculate_circle_area(5)
78.53975

>>> calculate_circle_area(10)
314.159

Note:
-----
This function assumes the input radius is positive.
Negative values will still calculate but may not represent real areas.


=== COMMENTING OUT CODE BLOCKS ===
The code block above is completely ignored

=== MODULE DOCSTRING EXAMPLE ===
Module-level documentation demonstrated above
```

5. Comment Best Practices

Writing good comments is an art that improves code quality and maintainability. Here are the essential guidelines for effective commenting.

In [8]:
```
# Comment Best Practices - Examples

print("=== GOOD vs BAD COMMENT EXAMPLES ===")

# BAD: Stating the obvious
x = 5  # Set x to 5
y = 10 # Set y to 10

# GOOD: Explaining the purpose or context
user_age = 25        # Minimum age required for account verification
max_attempts = 3     # Security limit for login failures

print(f"User age: {user_age}, Max attempts: {max_attempts}")

print("\n=== EXPLAIN WHY, NOT WHAT ===")

# BAD: Just describing what the code does
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [n for n in numbers if n % 2 == 0]  # Get even numbers

# GOOD: Explaining why this is needed
user_scores = [85, 92, 78, 96, 88, 91, 84, 89]
passing_scores = [score for score in user_scores if score >= 80]  # Filter scores

print(f"Passing scores: {passing_scores}")

print("\n=== GOOD COMMENTING PRACTICES ===")

def calculate_monthly_payment(principal, annual_rate, years):
    """
    Calculate monthly mortgage payment using the standard formula.

    Uses the formula: M = P * [r(1+r)^n] / [(1+r)^n - 1]
    where M = monthly payment, P = principal, r = monthly rate, n = total payments
    """
    # Convert annual rate to monthly decimal rate
    monthly_rate = annual_rate / 12 / 100

    # Calculate total number of payments
    total_payments = years * 12

    # Handle special case of 0% interest rate to avoid division by zero
    if monthly_rate == 0:
        return principal / total_payments

    # Apply the mortgage payment formula
    # This formula accounts for compound interest over the loan term
    numerator = monthly_rate * (1 + monthly_rate) ** total_payments
    denominator = (1 + monthly_rate) ** total_payments - 1
    monthly_payment = principal * (numerator / denominator)

    return monthly_payment

# Example usage with realistic values
home_price = 300000      # Home purchase price
```

```
down_payment = 60000     # 20% down payment
loan_amount = home_price - down_payment  # Remaining amount to finance
interest_rate = 3.5      # Annual interest rate (%)
loan_term = 30           # Loan term in years

payment = calculate_monthly_payment(loan_amount, interest_rate, loan_term)
print(f"Monthly mortgage payment: ${payment:.2f}")

print("\n=== COMMENT MAINTENANCE ===")

# Comments should be updated when code changes
def process_order(items, customer_type="regular"):
    """
    Process customer order and calculate total with appropriate discounts.

    Updated: Now includes premium customer handling (was previously VIP only)
    TODO: Add seasonal discount logic
    """
    base_total = sum(item['price'] for item in items)

    # Apply customer-specific discounts
    # Updated discount structure as of Q3 2024
    if customer_type == "premium":
        discount = 0.15  # 15% discount for premium customers
    elif customer_type == "vip":
        discount = 0.25  # 25% discount for VIP customers
    else:
        discount = 0.0   # No discount for regular customers

    total = base_total * (1 - discount)
    return total

# Example order processing
sample_items = [
    {"name": "Laptop", "price": 999.99},
    {"name": "Mouse", "price": 29.99}
]

regular_total = process_order(sample_items, "regular")
premium_total = process_order(sample_items, "premium")

print(f"Regular customer total: ${regular_total:.2f}")
print(f"Premium customer total: ${premium_total:.2f}")
```

```
=== GOOD vs BAD COMMENT EXAMPLES ===
User age: 25, Max attempts: 3

=== EXPLAIN WHY, NOT WHAT ===
Passing scores: [85, 92, 96, 88, 91, 84, 89]

=== GOOD COMMENTING PRACTICES ===
Monthly mortgage payment: $1077.71

=== COMMENT MAINTENANCE ===
Regular customer total: $1029.98
Premium customer total: $875.48
```

In [9]:
```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.
```

```
    Parameters:
    length (float): The length of the rectangle
    width (float): The width of the rectangle

    Returns:
    float: The area of the rectangle (length * width)

    Example:
    >>> calculate_area(5, 3)
    15
    """
    return length * width

# Example of good commenting
def process_student_grades(grades):
    """Process a list of student grades and return statistics."""

    # Filter out invalid grades (should be between 0-100)
    valid_grades = [grade for grade in grades if 0 <= grade <= 100]

    # Calculate average grade
    if valid_grades:  # Check if list is not empty to avoid division by zero
        average = sum(valid_grades) / len(valid_grades)
        return {
            'average': round(average, 2),
            'total_students': len(valid_grades),
            'highest': max(valid_grades),
            'lowest': min(valid_grades)
        }
    else:
        return {'error': 'No valid grades found'}

# Test the functions
area = calculate_area(5, 3)
print(f"Rectangle area: {area}")

sample_grades = [85, 92, 78, 96, 88, -5, 105, 91]  # Include invalid grades
stats = process_student_grades(sample_grades)
print(f"Grade statistics: {stats}")
```

```
Rectangle area: 15
Grade statistics: {'average': 88.33, 'total_students': 6, 'highest': 96, 'lowest':
78}
```

Key Takeaways

- **Single-line comments** use `#` and are great for brief explanations
- **Multi-line comments** use `"""` or `'''` for longer documentation
- **Docstrings** document functions, classes, and modules - accessible via `.__doc__`
- **Inline comments** should be used sparingly with proper spacing
- **Comment out code** temporarily using `#` for debugging
- **Explain WHY, not WHAT** - focus on the reasoning behind the code
- **Keep comments current** - update them when code changes

Comment Style Guide

DO:

- Write clear, concise comments that add value
- Explain complex algorithms and business logic
- Document function parameters and return values
- Use proper grammar and spelling
- Keep comments up-to-date with code changes
- Use comments to explain non-obvious code decisions

DON'T:

- Over-comment simple, self-explanatory code
- Write misleading or outdated comments
- Use comments to explain poorly written code (refactor instead!)
- Leave commented-out code in production
- Write comments that just repeat what the code does

DOCSTRING CONVENTIONS:

- Use triple double quotes `"""` for consistency
- Include a brief one-line summary
- Add detailed description for complex functions
- Document parameters, return values, and exceptions
- Provide usage examples when helpful

Practice Exercises

Try these to improve your commenting skills:

1. **Comment Review**: Take some uncommented code and add appropriate comments
2. **Docstring Practice**: Write comprehensive docstrings for functions
3. **Code Documentation**: Document a small project with module, class, and function docstrings
4. **Comment Refactoring**: Find over-commented code and improve it
5. **Debug Comments**: Practice using comments for debugging and testing

---

Course Information

Learn Python Programming from Scratch

Author: Prakash Ukhalkar

Topic: Python Fundamentals - Comments and Documentation

Built with ❤️ for the Python learning community