Notes
CSC-111
Intro to Programming

qxdeng1991

January 2025

Contents

1 Introduction

1.1 Identifiers

An **identifier** is the name used to identify variables, functions, classes, or other objects in Python. Identifiers must follow these rules:

- They can contain letters (A-Z, a-z), digits (0-9), and underscores (_).
- They must start with a letter or an underscore, not a digit.
- They are case-sensitive (e.g., `myVar` and `MyVar` are different).
- Reserved keywords cannot be used as identifiers.

Example of valid and invalid identifiers:

```
# Valid identifiers
variable1 = 10
_myVar = "Hello"
MyVar = 20

# Invalid identifiers
1variable = 30   # Cannot start with a digit
my-var = 40      # Hyphen is not allowed
if = 50          # 'if' is a keyword
```

1.2 Naming Traditions

Python supports several naming conventions for identifiers:

- Camel Case [Function]
  - Each word, except the first, starts with a capital letter
  - myVariableName = ”John”
- Pascal Case [Class & Structure]
  - Each word starts with a capital letter
  - MyVariableName = ”John”
- Snake Case [Variable]
  - Each word is separated by an underscore character
  - my_variable_name = ”John”

1.3 Difference Between Variables and Identifiers

- A **variable** is a symbolic name for a value that can be changed during program execution.
- An **identifier** is the name used to define variables, functions, or other entities.
- All variables are identifiers, but not all identifiers are variables.

```
# 'x' is an identifier for a variable
x = 10

# 'my_function' is an identifier for a function
def my_function():
    print("This is a function")
```

1.4 Keywords

- **Keywords** are reserved words in Python that have predefined meanings and cannot be used as identifiers. - Examples include `if`, `else`, `while`, `for`, `def`, `return`, etc. - To view all keywords in Python, use the `keyword` module:

```
import keyword
print(keyword.kwlist)
```

1.5 Comment your code

Comments are used to add explanations to your code and are ignored during execution.

- Single-line comments start with `#`:

```
# This is a single-line comment
print("Hello, World!")  # Inline comment
```

- Multi-line comments can be created using triple quotes:

```
"""
This is a multi-line comment.
It spans multiple lines.
"""
print("Multi-line comment example")
```

1.6 Indentation Is Important

Python uses indentation to define blocks of code, such as those in loops or functions. Indentation is mandatory and replaces the use of braces {} found in other languages. A **block of code** is a group of statements that belong together logically and are executed as a unit.
In Python, the start of a block is indicated by a colon (`:`), and all statements within the block are indented consistently.

```
if True:  # The colon marks the start of the block
    print("This is a block of code")  # Indented statement
    print("These statements belong to the same block")  # Indented statement
print("This is outside the block")  # Not indented
```

Explanation: - The two `print()` statements inside the `if` condition are part of the same block because they are indented to the same level. - The last `print()` statement is outside the block because it is not indented.

Again, indentation is mandatory in Python and ensures that the code structure is visually clear. Improper indentation will result in an **IndentationError** or unexpected behavior.

```
if True:
    print("Properly indented")
  print("Improperly indented")  # This will raise an error
```

Blocks can be nested, meaning that one block can contain another block of code. Each level of nesting requires additional indentation.

```
if True:  # Outer block
    print("Outer block starts")
    if False:  # Nested block
        print("Inner block")
    print("Outer block ends")
```

Python does not enforce a specific number of spaces for indentation, but **consistency** is crucial. Common conventions:

- Use 4 spaces for **each level of** indentation.

```
if True:
    print("4 spaces indentation")
    print("4 spaces indentation")
if False:
  print("2 spaces indentation")  # not consistant
  print("2 spaces indentation")  # not consistant
```

- Avoid mixing tabs and spaces, as this can lead to errors.

1.7 Multi-Line Statements

Python allows breaking a long statement into multiple lines using:

```
# A backslash \ at the end of the line:
total = 1 + 2 + 3 + \
        4 + 5
# Implicit line continuation inside parentheses, brackets, or braces:
total = (1 + 2 + 3 +
         4 + 5)
```

2 Operators

We start with two frequently used forms of numeric data, which focus on numbers without fractional parts:

- A **natural number** is any value in the set $\mathbb{N}$. We use the symbol $\mathbb{N}$ for natural numbers. In computer science, we often treat 0 as a natural number as well, $\mathbb{N} = \{0, 1, 2, 3, ...\}$
- An **integer** is any value in the set $\mathbb{Z}$. We use the symbol $\mathbb{Z}$ for integers, $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, 3, ...\}$

Naturally, the natural numbers are a subset of the integers: every natural number is an integer, but not every integer is a natural number. In Python, the `int` data type represents both natural numbers and integers. In Python, an `int` literal is just a sequence of digits, potentially preceded by a minus sign. For instance, `110` or `-3421`:

```
print(type(10))
>>><class 'int'>
```

2.1 Arithmetic Operations on Natural Numbers and Integers

All integers support the typical arithmetic operations: addition (+), subtraction (-), multiplication (*), and exponentiation (**). They also include division, though it is a particular case that will be covered in more detail later.

One arithmetic operation that you may find less familiar is **modulo**, which gives the remainder upon dividing one integer by another. We use the percent symbol `%` to denote the modulo operation, writing `x % y` to signify "the remainder when x is divided by y." For instance, `10 % 3 = 1` and `27 % 5 = 2`.

It should come as no surprise that Python offers these arithmetic operations, employing operators comparable to their mathematical counterparts:

```
>>> 2 + 3
5
>>> 2 - 5
-3
>>> -2 * 10
-20
>>> 2 ** 5  # This means "2 to the power of 5"
```

```
32
>>> 10 % 4
2
```

Additionally, Python respects the conventional precedence rules for arithmetic, such as exponentiation is computed before multiplication, and multiplication before addition or subtraction:

```
>>> 1 + 2 ** 3 * 5  # This is equivalent to "1 + ((2^3)*5)"
41
```

As in standard mathematics, extensive expressions like the example above can be difficult to read. Python permits the use of parentheses to clarify operations:

```
>>> 1 + ((2 ** 3) * 5)  # Same as the previous example
41
>>> (1 + 2) ** (3 * 5)  # A different grouping: "(1+2)^(3*5)"
14348907
```

Adding, subtracting, multiplying, or exponentiating two integers invariably results in an integer, so Python consistently yields an `int` in those instances. However, dividing two integers may not result in an integer at all. While math accommodates fractions, Python needs a specific approach.

Python actually provides two types of division operators:

1. The `//` operator, known as **floored division** (or integer division). For two integers `x` and `y`, `x // y` corresponds to the fraction $\frac{x}{y}$ rounded down to the nearest integer. This is the *quotient* of dividing `x` by `y`.

```
>>> 6 // 2
3
>>> 15 // 2  # 15 / 2 = 7.5, which rounds down to 7
7
>>> -15 // 2  # -15 / 2 = -7.5, which rounds down to -8
-8
```

2. The `/` operator, sometimes called **exact division**, which can yield fractional results.

```
>>> 15 / 2
7.5
```

   In such circumstances, Python produces a value of a different data type called `float`, which represents real numbers (including fractions).

The table below also show the shorthand assignment operators:

Operator	Example	Equivalent To	Description
`=`	`x = 10`	`x = 10`	Assigns a value to a variable.
`+=`	`x += 5`	`x = x + 5`	Adds the right-hand value to the variable on the left.
`-=`	`x -= 5`	`x = x - 5`	Subtracts the right-hand value from the variable on the left.
`*=`	`x *= 5`	`x = x * 5`	Multiplies the variable on the left by the right-hand value.
`/=`	`x /= 5`	`x = x / 5`	Divides the variable on the left by the right-hand value.
`//=`	`x //= 5`	`x = x // 5`	Performs integer (floored) division and assigns the result.
`%=`	`x %= 5`	`x = x % 5`	Computes the remainder (modulo) and assigns the result.
`**=`	`x **= 5`	`x = x ** 5`	Raises the variable on the left to the power of the right-hand value.

2.2 Fractional and Real Numbers

Next, let's explore non-integer numbers. You may recall several familiar number sets from your prior math studies:

- A **rational number** is any value in $\mathbb{Q}$, representing possible fractions. This includes values like $\frac{1}{2}$ and $\frac{5}{4}$, as well as integers (since, for example, $2 = \frac{2}{1}$). We use $\mathbb{Q}$ for the set of rational numbers.
- An **irrational number** is a number with an infinite, non-repeating decimal expansion, such as $\sqrt{2}$, π, or e. We often use symbols like $\mathbb{R} \setminus \mathbb{Q}$ (the reals without the rationals) to represent the irrationals.
- A **real number** is either rational or irrational, denoted by $\mathbb{R}$.

Python provides a separate data type, `float`, for representing non-integer values. A `float` literal is expressed by a sequence of digits, a decimal point (`.`), then more digits. For example:

```
>>> 7.5
7.5
>>> .123
0.123
>>> -1000.00000001
-1000.00000001
```

2.2.1 Operations on Fractional and Real Numbers

Mathematically, the same arithmetic and comparison operators we saw for `int` values apply to `float` values. Here are some examples:

```
>>> 3.5 + 2.4
5.9
>>> 3.5 - 20.9
-17.4
>>> 3.2 > 1.5
True
>>> -3.2 > -1.5
False
>>> 3.5 / 2.5
1.4
```

2.2.2 The Limitations of `float`

Consider the following scenario. The square root of a number x is $x^{1/2}$. Let us see how Python approximates $\sqrt{2}$:

```
>>> 2 ** 0.5
1.4142135623730951
```

Notice the challenge: $\sqrt{2}$ is an irrational number with an infinite, non-repeating decimal representation. However, the Python interpreter is constrained by finite memory, so it cannot fully represent $\sqrt{2}$. More specifically, computers use binary (0s and 1s) to encode all data, including numbers, and only have a finite amount of memory, making an exact representation of $\sqrt{2}$ impossible. Hence, the `float` value 1.4142135623730951 is only an approximation. Let's square it:

```
>>> 1.4142135623730951 ** 2
2.0000000000000004
```

And more dramatically:

```
>>> (2 ** 0.5) ** 2 == 2
False
```

This reveals a key limitation: a `float` approximates real numbers, and, except in special cases (like 2.5), it does not capture them exactly. This is inevitable due to the finite nature of computer memory. Instead of a == b, use:

```
import math

def almost_equal(a, b, tol=1e-9):
    return abs(a - b) < tol

print(almost_equal(0.1 + 0.2, 0.3))  # True
```

This method compares the difference against a small threshold (tol).

2.2.3 Mixing `int` and `float`

If an arithmetic operation receives one `int` and one `float`, it yields a `float`. Even if only a single float is involved in a longer expression, the entire expression ends as a `float`, including scenarios in which the final mathematical answer is an integer:

```
>>> 12 - 4 * 5 // (3.0 ** 2) + 100
110.0
```

2.3 Booleans and Comparisons

When comparing two numbers, we typically utilize symbols for equality or inequality (e.g., $=$ or $\neq$), along with symbols like $>$, $<$, $\geq$, and $\leq$ to indicate relative size. As with arithmetic, Python offers corresponding operators for these comparisons:

Operation	Description
`a == b`	Determines if `a` and `b` are equal
`a != b`	Determines if `a` and `b` are not equal
`a > b`	Determines if `a` is greater than `b`
`a < b`	Determines if `a` is less than `b`
`a >= b`	Determines if `a` is greater than or equal to `b`
`a <= b`	Determines if `a` is less than or equal to `b`

```
>>> 4 == 4
True
>>> 4 != 6
True
>>> 4 < 2
False
>>> 4 >= 1
True
```

A **boolean** is a value drawn from the set {True, False}. You can think of a boolean value as the answer to a Yes/No question. We have also encountered booleans in the preceding section, where they appeared as the result of comparison operations. For instance, when we write `3 < 5`, the value of this expression is not a number but a boolean (`True` in this example).

In Python, boolean data is represented with the `bool` data type. In contrast to the wide range of numeric values, there are only two literal values of type `bool`: `True` and `False`.

Booleans can be combined using several operations, with the three most common being:

- **not**: Reverses the value of a boolean. `not True` is `False`, and `not False` is `True`.
- **and**: Takes two boolean values and results in `True` only if both values are `True`, and `False` otherwise. For instance, `True and False` is `False`, while `True and True` is `True`.
- **or**: Takes two boolean values and yields `True` if at least one of the values is `True`, and `False` otherwise. For example, `True or False` is `True`, while `False or False` is `False`.

In Python, these three logical operations match up with three corresponding operators, which are written out as English words rather than symbols: `not`, `and`, and `or`. Unlike arithmetic and comparison operators, which are often

symbols (like + and >), Python uses words for logical operations to promote code readability. Consider the examples below:

```
>>> not True
False
>>> True and True
True
>>> True and False
False
>>> False or True
True
>>> False or False
False
```

Similar to how arithmetic expressions can be nested, logical operator expressions can also be nested. We can even mix in arithmetic comparison operations:

```
>>> True and (False or True)
True
>>> (3 == 4) or (5 > 10)  # Both (3 == 4) and (5 > 10) evaluate to False
False
```

Each of `not`, `and`, `or` has its own level of **precedence**, which determines the order in which Python evaluates them when no parentheses are used. Specifically, Python follows the rule:

$$\texttt{not} > \texttt{and} > \texttt{or}.$$

Let's see a more involved example combining `not`, `and`, and `or` without parentheses:

```
True or not True and False
```

In this expression:

- Step 1: Evaluate `not True` → `False`.
- Step 2: Evaluate `False and False` → `False`.
- Step 3: Evaluate `True or False` → `True`.

Thus, the entire expression produces `True`.

By introducing parentheses, we can alter the evaluation order. Compare:

```
# Without parentheses
True or not True and False
# parent steps: (not True) -> False, then (False and False) -> False, finally True or False -> True

# With parentheses
(True or not True) and False
# parent steps: (True or not True) -> True, then True and False -> False

```

Try some expressions by yourselves:

```
True or False and False
(True or False) and False
not False and True or False
```

2.4 Binary and Hexadecimal

The **decimal system** is the one we use in everyday life. It is called *base 10* because each digit represents a power of 10, depending on its position. For example, the decimal number 198 means:

$$1 \times 10^2 + 9 \times 10^1 + 8 \times 10^0.$$

2.4.1 Binary System (Base 2)

The **binary system** uses only two digits: 0 and 1. It is called *base 2* because each digit represents a power of 2, depending on its position from the right (starting at 0). For example, the binary number 1011_2 means:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11.$$

This is why 1011_2 corresponds to the decimal number 11.

To convert a decimal (base 10) number into binary (base 2), one common method is:

1. Divide the decimal number by 2.
2. Record the remainder (0 or 1).
3. Take the quotient and repeat the process until the quotient becomes 0.
4. The binary number is the sequence of remainders read *backwards* (from last to first).

Example: Convert 13_{10} to binary.

$$\begin{aligned} 13 \div 2 &= 6 \quad \text{remainder } 1, \\ 6 \div 2 &= 3 \quad \text{remainder } 0, \\ 3 \div 2 &= 1 \quad \text{remainder } 1, \\ 1 \div 2 &= 0 \quad \text{remainder } 1. \end{aligned}$$

Reading the remainders from bottom to top, the binary representation is $\mathbf{1101_2}$.

To convert a binary (base 2) number to decimal (base 10), we can sum up the powers of 2 for each position where the binary digit is 1.

Example: Convert 1101_2 to decimal.

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13.$$

Hence, $1101_2 = 13_{10}$.

2.4.2 Hexadecimal System (Base 16)

The **hexadecimal system** uses sixteen distinct digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

- Digits 0–9 represent the values 0 to 9.
- Letters A–F represent the values 10 to 15.

Being *base 16*, each position represents a power of 16. For example, the hexadecimal number $\texttt{1A3}_{16}$ means:

$$1 \times 16^2 + 10 \times 16^1 + 3 \times 16^0 = 256 + 160 + 3 = 419.$$

To convert a decimal number to hexadecimal:

1. Divide the decimal number by 16.
2. Record the remainder (0–9 or A–F).
3. Continue dividing the quotient by 16 until it becomes 0.
4. Collect the remainders from last to first to form the hexadecimal number.

Example: Convert 254_{10} to hexadecimal.

$$\begin{aligned} 254 \div 16 &= 15 \quad (\text{remainder } 14) \to E, \\ 15 \div 16 &= 0 \quad (\text{remainder } 15) \to F. \end{aligned}$$

Reading the remainders from bottom to top: $\mathbf{FE_{16}}$. Thus, $254_{10} = FE_{16}$.

To convert from hexadecimal to decimal, multiply each digit by 16^k, where k is the position index from the right, starting at 0.

Example: Convert $\texttt{FE}_{16}$ to decimal.

$$F \times 16^1 + E \times 16^0 = 15 \times 16 + 14 \times 1 = 240 + 14 = 254.$$

2.5 Bitwise Operations

In Python, `bitwise operators` allow you to manipulate individual bits within integer values. Python provides the following bitwise operators:

- `&` — **Bitwise AND**
- `|` — **Bitwise OR**
- `^` — **Bitwise XOR**
- `~` — **Bitwise NOT** (complement)
- `<<` — **Left Shift**
- `>>` — **Right Shift**

Bitwise operators operate on the binary representations of integers. For instance, consider the decimal value `6`:

$$6_{10} = 110_2 \quad \text{(in binary)}$$

Similarly, the decimal value `3`:

$$3_{10} = 011_2 \quad \text{(in binary, 3 bits for illustration)}$$

When applying bitwise operators, Python effectively converts the integers to their binary form, performs the operation bit by bit, then converts back to a Python integer.

2.5.1 Bitwise AND (`&`)

The **Bitwise AND** operator `&` takes two integer operands and performs the logical *AND* on each corresponding pair of bits. The result is `1` only if both bits are `1`, otherwise `0`.

```
# Decimal values 6 and 3
x = 6  # binary: 110
y = 3  # binary: 011

result = x & y  # bitwise AND
print(result)
# Explanation:
# 6 (110 in binary)
# 3 (011 in binary)
# 110 & 011 = 010 (binary) = 2 (decimal)
```

The output will be `2`.

2.5.2 Bitwise OR (`|`)

The **Bitwise OR** operator `|` compares each pair of bits from two integers. If at least one bit is `1`, the result is `1`; otherwise, `0`.

```
x = 6  # binary: 110
y = 3  # binary: 011

result = x | y
print(result)
# Explanation:
# 6 (110 in binary)
# 3 (011 in binary)
# 110 | 011 = 111 (binary) = 7 (decimal)
```

The output will be `7`.

2.5.3 Bitwise XOR (`^`)

The **Bitwise XOR** (`^`) operator yields `1` if exactly one bit of the pair is `1`, otherwise it produces `0`. Another way to remember XOR: it returns `1` if the bits differ and `0` if they are the same.

```python
x = 6  # 110
y = 3  # 011

result = x ^ y
print(result)
# Explanation:
# 6 (110 in binary)
# 3 (011 in binary)
# 110 ^ 011 = 101 (binary) = 5 (decimal)
```

The output will be `5`.

2.5.4 Bitwise NOT (~)

The **Bitwise NOT** (`~`) operator flips (inverts) each bit of its single operand. In typical 8-bit representation, `~01010110` would become `10101001`. However, Python integers are not limited to 8 bits; they have arbitrary length.

Internally, Python represents negative numbers using *two's complement* with an infinite bit-length concept. For any integer `x`, applying `~x` (bitwise NOT) is effectively:

$$\sim x = -x - 1.$$

For example:

$$\sim 5 = -6, \quad \sim (-2) = 1.$$

```python
x = 5
result = ~x
print(result)  # -6

y = -2
print(~y)      # 1
```

2.5.5 Left Shift (<<) and Right Shift (>>)

The **left shift** operator `<<` shifts the bits of its left operand to the left by the number of positions specified by the right operand. New bits on the right side are filled with `0`s. For positive integers, a left shift by `n` positions is equivalent to multiplying by 2^n.

```python
x = 3        # 011 in binary
shifted = x << 2
print(shifted)
# Explanation:
# 3 (011) << 2 => 01100 (binary) = 12 (decimal)
```

The **right shift** operator `>>` shifts bits to the right by the specified number of positions. Bits that shift off the right end are discarded. For **positive integers**, bits on the left end are filled with `0`s, effectively performing a floor division by 2^n.

```python
x = 13       # 1101 in binary
shifted = x >> 2
print(shifted)
# Explanation:
# 13 (1101) >> 2 => 0011 (binary) = 3 (decimal)
```

The output will be `3`.

You can combine multiple bitwise operations in a single expression. For instance:

```python
x = 12        # binary: 1100
y = 5         # binary: 0101

result = (x & y) << 1
print(result)
# Steps:
# x & y => 1100 & 0101 = 0100 (4 decimal)
# then 4 << 1 => 1000 (binary) = 8 decimal
```

2.5.6 Negative integers and its shift

For negative numbers, Python uses a **two's complement**. In a typical N-bit system:

- A positive integer `x` is stored as its normal binary form.
- A negative integer `-x` is stored by:
  1. Taking the binary representation of `x`.
  2. Inverting all bits (*one's complement*).
  3. Adding 1 (*two's complement*).

For instance, in an 8-bit world:

$$+6 \equiv 00000110, \quad -6 \equiv 11111010.$$

Here, -6 results from inverting 00000110 to get 11111001, then adding $1 \rightarrow 11111010$.

However, **Python** doesn't limit itself to 8 bits (or any fixed width). Internally, it uses as many bits as needed plus extra sign bits for negative values. Conceptually, it's the same idea as two's complement on an infinite-length bit string.

When you perform `-6 >> 1`, Python does an `arithmetic shift`:

1. -6 in binary (two's complement style, infinitely extended) has a leading `1` bit to indicate it's negative.
2. Shifting right by 1 position duplicates that `1` bit on the left, ensuring the number remains negative.
3. Arithmetic shifting to the right by 1 is conceptually like dividing by 2 and *rounding downward* (floor).

Indeed,

$$-6 \div 2 = -3 \quad \text{(integer division, rounding down)},$$

so `-6 >> 1` evaluates to `-3`.

To explain this, let's assume an 8-bit system:

$$-6 \equiv 11111010 \quad \text{(2's complement, 8-bit)}.$$

Shifting right by 1 (arithmetic shift) means:

- Drop the rightmost bit.
- Shift everything right.
- **Fill in** the new leftmost bit with 1 (the sign bit).

So,

$$11111010 \gg 1 \;=\; 11111101,$$

which corresponds to -3 in 8-bit two's complement. To verify it:

1. **Identify the sign bit:**
   The most significant (leftmost) bit is `1`, indicating a **negative** number in two's complement (8-bit context).
2. **Invert all bits (One's Complement):**

   $$\texttt{11111101} \xrightarrow{\text{invert}} \texttt{00000010}.$$

   Flip each `1` to `0` and each `0` to `1`.
3. **Add 1 (Complete Two's Complement):**

   $$\texttt{00000010} + \texttt{00000001} = \texttt{00000011}.$$

   Now we have $\texttt{00000011}_2$, which equals 3 in decimal.

4. **Interpret the Result:**
   Because our original number was determined to be negative (from Step 1), we conclude

$$\texttt{11111101}_2 = -3 \text{ (decimal)}.$$

2.5.7 Bitmasking and Flags

A common usage of bitwise operations is in **bitmasking**, where each bit in a variable (often called a *mask*) represents a particular feature or flag. For example:

- `1 << 0` (0001) could represent a read permission.
- `1 << 1` (0010) could represent a write permission.
- `1 << 2` (0100) could represent an execute permission.

By `OR`ing these together, you can combine permissions. By `AND`ing with a mask, you can check if a particular bit is set.

3 Namespaces and Variable Scoping

In Python, a **variable** is a name that refers to a value stored in memory. Variables let you assign descriptive names to intermediate results in your programs, making your code easier to write, read, and maintain. The syntax for creating a variable in Python is:

```
<variable> = <expression>
```

where:

- `<variable>` is the name of the variable (a valid identifier) you want to create or update.
- `<expression>` is any piece of Python code that evaluates to a value.

For example, to store the result of a mathematical expression in a variable named `distance1`, you might write:

```
distance1 = ((2 - 1) ** 2 + (5 - 3) ** 2) ** 0.5
```

An **expression** produces a value when evaluated. In contrast, an **assignment statement** does not directly produce a value; instead, it associates a variable name with the value of an expression. When Python encounters an assignment statement like

```
distance1 = ((2 - 1) ** 2 + (5 - 3) ** 2) ** 0.5
```

it performs two steps:

1. **Evaluate the expression on the right-hand side (RHS) of the equals sign.** In this case, Python computes $\sqrt{(2-1)^2 + (5-3)^2}$.
2. **Bind the resulting value to the variable on the left-hand side (LHS).** The variable `distance1` will then refer to the numeric value produced by step 1.

After this assignment, you can use `distance1` as an expression that yields the stored value:

```
>>> distance1
2.23606797749979
```

3.1 Choosing Good Variable Names

As programs grow in complexity, it's crucial to use clear, descriptive variable names. Here are some guidelines:

1. **Use only lowercase letters, digits, and underscores** (_). For instance, `distance1`, not `Distance1`.
2. **Separate multiple words with underscores.** Write `total_distance` instead of `totaldistance` or `totalDistance`.
3. **Avoid single-letter variable names**, except in simple, mathematical contexts. While `d1` for `distance1` might be acceptable if you're closely following a math formula, names like `td` can be confusing if they don't convey meaning.
4. **Avoid non-standard acronyms or abbreviations.** Ensure that a new collaborator on your code can guess what each variable name represents.

For example:

```
total_distance = distance1 + distance2
```

is more descriptive than something like `td = d1 + d2`, which could be unclear.

3.2 Namespaces and the LGB Rule

When you reference a variable in Python, the interpreter must figure out which *specific* variable you're referring to. To prevent naming conflicts between your own variables and those in Python's standard library or in any third-party modules, Python uses the concept of **namespaces**. Essentially, **namespaces allow multiple parts of a program (or multiple programs) to safely reuse the same variable names without overwriting each other**.

A **namespace** is a mapping from variable names to objects. Python maintains different namespaces for different scopes in your program, and it follows a rule known as **LGB** when resolving variable names:

$$\text{Local} \rightarrow \text{Global} \rightarrow \text{Built-in}.$$

1. **Local Namespace**:
   - Each function invocation, class definition, or method in Python creates its own local namespace.
   - Variables assigned within a function belong to that function's local namespace.
2. **Global Namespace**:
   - The global namespace contains variables you define at the *top-level* of a script or an interactive session.
   - When you are not inside a function or class, variable assignments go to the global namespace.
3. **Built-in Namespace**:
   - Python automatically provides a number of built-in objects (e.g., `abs`, `len`, `Exception`, `range`).
   - These live in the built-in namespace, which is searched *last* if a variable name is not found in local or global namespaces.

When Python encounters a variable, it first looks in the *local* namespace. If not found, it checks the *global* namespace. Finally, if still not found, Python looks into its *built-in* namespace. If the variable is not found in any of these, Python raises a `NameError` exception.

Consider the following code:

```
x = 10  # Global variable

def foo():
    x = 5  # Local variable (shadows the global x)
    print("Local x:", x)

foo()
print("Global x:", x)

>>>Local x: 5
>>>Global x: 10
```

3.3 `import` Statement and Module Namespaces

Python uses the `import` statement to grant controlled access to *other* namespaces (such as modules). This avoids naming conflicts between your variables and any objects inside the module. The simplest form of the `import` statement is:

```
import module_name
```

This statement:

- Loads the code in `module_name` (if not already loaded).

- Creates a *module object* in the current namespace under the name `module_name`.
- You then refer to objects within that module using a *two-level* name, e.g. `module_name.object`.

```
import string

result = string.split('Welcome to the Ministry of Silly Walks')
print(result)
# ['Welcome', 'to', 'the', 'Ministry', 'of', 'Silly', 'Walks']
```

In this example, we *only* brought the `string` module's name into the local namespace. We still need to use `string.split(...)` to access the `split` function.

A more specific approach:

```
from module_name import some_func, another_func
```

This directly imports `some_func` and `another_func` into your local namespace. As a result, you can call these functions by name alone, without the module prefix:

```
from string import split, join

words = split('Welcome to the Ministry of Silly Walks')
print(words)
# ['Welcome', 'to', 'the', 'Ministry', 'of', 'Silly', 'Walks']
```

This approach can save typing and is efficient if you only need a few objects from the module. **You need to be aware of potential name conflictions.**

In some situations, you might want everything from the module's namespace to be directly available in your local namespace:

```
from string import *
```

Now, all functions, classes, and constants from `string` become available without prefix. **Use caution** with this method:

- Names in the module might override the names of your own variables.
- It imports many names, potentially slowing down `dir()` lookups or creating name collisions.

A more popular approach:

```
import <module> as <alias>
```

lets you import an entire module under a shorter or more convenient name (the "alias"). This can make your code cleaner and easier to maintain, especially if a module's name is long, or if you want to avoid naming conflicts with other modules or variables.

```
import math as m
# Using math.sqrt:
x = m.sqrt(25)   # x = 5.0

# Using math.sin, math.pi:
y = m.sin(m.pi / 2)  # y = 1.0
```

Here, the `math` module is imported and given the name `m`. Instead of writing `math.sqrt(...)`, you can simply write `m.sqrt(...)` throughout your code.

- **Clarity**: If a module name is long (e.g., `pandas`), you can shorten it to something more concise (e.g., `pd`), which is a standard alias in the Python data science community.
- **Avoiding Conflicts**: If your code has a variable named `time`, you might want to avoid `import time`. Instead, you could do `import time as time_module`.
- **Readability & Conventions**: Some libraries in Python have well-known community conventions for their alias names:

  `import numpy as np`, `import pandas as pd`, `import matplotlib.pyplot as plt`

  This makes it easier for others to read and understand your code.

4 Input and Output

In Python, the simplest way to print something to the console is by using the `print` function.

```
print("Hello, World!")

# You can also print multiple items separated by spaces:
print("Name:", "Alice", "Age:", 25)

# By default, print adds a newline at the end.
# To change the separator or end character, you can pass keywords:
print("Hello", "World", sep=" - ", end=" *** ")
print("This is the next line")
```

In the example above:

- `sep=" - "` changes the separator between items.
- `end=" *** "` changes what is appended after the last item (instead of the default newline).

In Python, you can read input from the console using the `input()` function. By default, `input()` returns a string.

```
user_name = input("Enter your name: ")
print("Hello,", user_name)
```

You can prompt the user with a string (e.g., `"Enter your name: "`). **Whatever the user types will be returned as a string**. If you need a numeric value, you must explicitly convert it:

```
user_age_str = input("Enter your age: ")
user_age = int(user_age_str)  # Convert to int
print("You will be", user_age + 1, "next year!")
```

Alternatively, you can write:

```
user_age = int(input("Enter your age: "))
```

4.1 Formatting Your Output

You can control how output appears using string formatting techniques:

- **f-strings (Python 3.6+)**
- `format` method
- **old-style % formatting** (less recommended for new code)

4.1.1 Using f-strings

```
name = "Alice"
age = 25
print(f"My name is {name}, and I am {age} years old.")
```

Here, {name} and {age} are replaced with the values of those variables.

4.1.2 Using the format method

```
name = "Bob"
age = 30
print("My name is {}, and I am {} years old.".format(name, age))
```

4.1.3 Old-style % formatting

Formatting was commonly done using the % operator. Here's a quick demonstration of how it works:

```
name = "Alice"
age = 25
pi = 3.141592653589793

# 1) Using %s for strings and %d for integers
print("My name is %s and I am %d years old." % (name, age))
# Output: My name is Alice and I am 25 years old.

# 2) Using %f for floats
print("The value of pi is approximately %f." % pi)
# Output: The value of pi is approximately 3.141593.

# 3) Specifying width and precision
print("Pi truncated to 3 decimal places: %.3f" % pi)
# Output: Pi truncated to 3 decimal places: 3.142

# 4) Combining different types
print("Name: %s, Age: %d, Pi: %.2f" % (name, age, pi))
# Output: Name: Alice, Age: 25, Pi: 3.14
```

- %s for strings
- %d (or %i) for integers
- %f for floats
- %.nf to format floats to n decimal places

Multiple placeholders can be provided in a tuple after the % symbol, like:

```
"Some text %s and %s" % ("first", "second")
```

If you need to include literal % characters, escape them as %%.

5 Numbers

5.1 Scientific Notation

Python supports scientific notation to represent very large or very small numbers. The letter e or E is used to denote the power of 10.

```
x = 1.23e4  # Equivalent to 1.23 * 10^4
y = 5.67E-3  # Equivalent to 5.67 * 10^-3
print(x)  # Output: 12300.0
print(y)  # Output: 0.00567
```

5.2 Immutability of Numbers

In Python, numbers are immutable objects. **This means that any operation on a number creates a new object rather than modifying the existing one.**

- `id()` is a built-in Python function that returns the memory address of the object a variable refers to.
- In some contexts, Python may optimize memory by reusing the same numerical object for different variables if they have the same value. This is a runtime optimization and depends on whether Python detects the values as identical during assignment within the same scope.

```
#number
#number
var1 = 1
var2 = 10
var3 = 10
# id() show variable's memory address
print(id(var1), id(var2))
print(id(var2), id(var3))
var3 += 5
print(id(var3))


>>>4341770544 4341770832
>>>4341770832 4341770832
>>>4341770992
```

However, if you create the objects in a different way or across different execution scopes, Python might not reuse the object:

```
var1 = 10000
var2 = int("10000")
print(id(var1), id(var2))

>>>4407893584 4378237200
```

5.3 Type Conversion

Python provides built-in functions to convert between different numerical types:

- `int()` converts to an integer.
- `float()` converts to a floating-point number.

```
x = 3.14
print(int(x))  # Output: 3
print(float(5))  # Output: 5.0
```

5.4 The `math` Module

The `math` module provides advanced mathematical functions and constants:

- Constants like `math.pi` and `math.e`.
- Functions for trigonometry, logarithms, square roots, and more.

Example:

```
#some math functions
import math
print(math.ceil(10.3)) #returns the upper integer of the number
print(math.floor(4.9)) #returns the rounded integer of the number
print(math.exp(1)) #returns e to the power of x
print(math.fabs(10)) #returns the absolute value of the number as a floating point number
print(math.log(math.e), math.log(100,10)) #returns the logarithm of x based on e or specified value
print(math.log10(100)) #returns the logarithm of x based on 10
print(math.pow(2,3)) #The value after x**y operation.
print(math.sqrt(4)) #returns the square root of the number x.
print(math.pi) #pi Mathematical constant pi
print(math.e) #e Mathematical constant e, e is the natural constant (natural constant).
```

Python can handle special numerical values such as infinity and NaN (Not a Number). These are available in the `math` module:

```
import math

print(math.inf)  # Positive infinity
print(-math.inf)  # Negative infinity
print(math.nan)  # Not a Number

# Checking special values
print(math.isinf(math.inf))  # Output: True
print(math.isnan(math.nan))  # Output: True
```

5.5 Built-in Functions

Python provides several built-in functions for numerical operations:

- `max()` and `min()` to find the maximum and minimum values.
- `round()` to round numbers.
- `abs()` to find the absolute value.

Example:

```
nums = [3, 7, -2, 8]
print(max(nums))  # Output: 8
print(min(nums))  # Output: -2
print(round(3.14159, 2))  # Output: 3.14
print(abs(-5))  # Output: 5
```

5.6 The `random` Module

The `random` module is used for generating random numbers.

- The random.seed() function initializes the random number generator. If you use the same seed value (e.g., 2) and run the code multiple times, you will get the same sequence of random numbers. Commenting this out allows the random number generator to use a system-based seed, producing different results on each run.
- randint(a, b), a and b are all **inclusive**!(This is a historical issue.) It is the only function that includes the upper bound (b) in its range of possible outputs.

```
#random functions
import random
# random.seed(2) # try to change it, run your code several times and see what happened, this is for reproducibility!
print(random.choice([1,2,3,4,5,6,7,8,9]))
```

```
print(random.randrange(1,100,10)) # it only randomly pick from 1, 11, 21, 31, 41, 51, 61, 71, 81, 91
print(random.random()) #generates a random floating-point number between 0.0 and 1.0.
var1 = [1,2,3,4,5,6,7,8,9]
random.shuffle(var1)#in place operation
print(random.uniform(1, 10)) #random.uniform(a, b) generates a random floating-point number between a and b.
print(random.randint(1, 6))
```

5.7 Decimal Mode

The `decimal` module offers precision for decimal arithmetic, suitable for financial calculations. This method achieves a higher accuracy, but cost more computation resources and memory:

```
from decimal import Decimal

# Using Decimal for precise calculation
sum = Decimal('0.0')
for i in range(10):
   sum += Decimal('0.1')

# Check if sum is equal to 1.0
print("Sum is:", sum)
print("Is sum exactly equal to 1.0?", sum == Decimal('1.0'))
```

5.8 Fraction Mode

The `fractions` module enables arithmetic with rational numbers:

```
from fractions import Fraction

# Creating fractions
frac1 = Fraction(1, 3)  # Fraction 1/3
frac2 = Fraction(2, 5)  # Fraction 2/5

# Arithmetic operations
sum_frac = frac1 + frac2          # Addition
difference_frac = frac1 - frac2   # Subtraction
product_frac = frac1 * frac2      # Multiplication
division_frac = frac1 / frac2     # Division

# Display the results
print("Fraction 1:", frac1)
print("Fraction 2:", frac2)
print("Sum:", sum_frac)
print("Difference:", difference_frac)
print("Product:", product_frac)
print("Division:", division_frac)
```

6 String

6.1 Creating a String

Strings in Python can be created using single quotes, double quotes, or triple quotes (for multi-line strings):

```
single_quote_string = 'Hello'
double_quote_string = "World"
triple_quote_string = '''This is
a multi-line string.'''
print(single_quote_string, double_quote_string, triple_quote_string)
```

6.2 String Operators

Python provides several operators for strings:

- **Concatenation**: Combine strings using the `+` operator.
- **Repetition**: Repeat strings using the `*` operator.
- **Membership**: Check for substring existence using `in` and `not in`.
- **Comparison**: Compare strings lexicographically using `==`, `!=`, `<`, `>`, etc.

Example:

```
s1 = "Hello"
s2 = "World"
print(s1 + s2)  # Concatenation: HelloWorld
print(s1 * 3)   # Repetition: HelloHelloHello
print("H" in s1)  # Membership: True
print(s1 < s2)  # Comparison: True (lexicographic order)
```

6.3 Slicing and Indexing

Strings in Python are indexed from 0. Negative indexing starts from the end of the string. Slicing extracts a substring:

```
s = "Python"
print(s[0])       # First character: P
print(s[-1])      # Last character: n
print(s[1:4])     # Substring from index 1 to 3: yth
print(s[:3])      # Substring from start to index 2: Pyt
print(s[3:])      # Substring from index 3 to end: hon
var1 = "abcdefghij"
print(var1[::2])  # Substring with step 2: acegi
print(var1[::-1]) # Reverse string: jihgfedcba
print(var1[:-1])  # All except the last character: abcdefghi
```

6.4 Escape Characters

Escape characters are used to include special characters in strings. Common escape sequences:

Escape Characters	Description
\\	Backslash
\n	Newline
\r	Carriage return
\t	Tab
\b	Backspace
\"	Double quote
\’	Single quote

```
#Escape Characters
print('\'Hello, world!\' \\')         #'Hello, world!' \
print("Hello, world!\nHow are you?")  # Hello, world!
                                      # How are you?
print("Hello, world!\rHow are you?")  # How are you?
print("Hello, world!\tHow are you?")  # Hello, world!        How are you?
print("Hello,\b world!")              # Hello world!
```

6.5 ASCII Table

The ASCII table maps characters to numerical values. Python provides functions to convert between characters and their ASCII values:

```
print(ord("A"))  # Output: 65 (ASCII value of A)
print(chr(97))    # Output: a (character for ASCII 97)
print("\x24")    #£
print("\x50\x79\x74\x68\x6f\x6e") # Python
```

6.6 Built-in Functions

Python includes numerous built-in functions for string manipulation:

```
# count() - Counts the occurrences of a substring
text = "Python is fun. Python is powerful."
print("count():", text.count("Python"))  # Output: 2

# find() - Finds the first occurrence of a substring
text = "Find the position of Python."
print("find():", text.find("Python"))  # Output: 20

# isdigit() - Checks if all characters are digits
number = "12345"
print("isdigit():", number.isdigit())  # Output: True

# isalnum() - Checks if all characters are alphanumeric
text = "Python3"
print("isalnum():", text.isalnum())  # Output: True

# join() - Joins elements of an iterable into a string
words = ["Python", "is", "awesome"]
print("join():", " ".join(words))  # Output: "Python is awesome"

# lower() - Converts all characters to lowercase
text = "Python IS Great!"
print("lower():", text.lower())  # Output: "python is great!"

# upper() - Converts all characters to uppercase
text = "Python is great!"
print("upper():", text.upper())  # Output: "PYTHON IS GREAT!"

# replace() - Replaces occurrences of a substring
text = "I like Java. Java is powerful."
print("replace():", text.replace("Java", "Python"))  # Output: "I like Python. Python is powerful."

# split() - Splits a string into a list
text = "Python,is,fun"
print("split():", text.split(","))  # Output: ['Python', 'is', 'fun']

# strip() - Removes leading and trailing whitespaces
text = "  Hello, Python!  "
print("strip():", text.strip())  # Output: "Hello, Python!"

# ljust() - Pads the string to the left
text = "Python"
print("ljust():", text.ljust(10, "-"))  # Output: "Python----"

# rjust() - Pads the string to the right
print("rjust():", text.rjust(10, "-"))  # Output: "----Python"

# center() - Centers the string
print("center():", text.center(10, "-"))  # Output: "--Python--"
```

Some other very interesting global functions:

```
# Demonstrating len(), str(), eval(), and exec()

# len() - Returns the length of an object
text = "Python"
numbers = [1, 2, 3, 4, 5]
print("len() on a string:", len(text))  # Output: 6
print("len() on a list:", len(numbers))  # Output: 5

# str() - Converts an object to a string
number = 12345
floating_point = 3.14
print("str() on an integer:", str(number))  # Output: "12345"
print("str() on a float:", str(floating_point))  # Output: "3.14"

# eval() - Evaluates a string expression
expression = "3 + 5 * 2"
print("eval() on an expression:", eval(expression))  # Output: 13

# exec() - Executes a block of code
code = """
def greet():
    print("Hello from exec!")

greet()
"""
print("exec() to execute code:")
exec(code)  # Output: "Hello from exec!"
```

6.7 Immutability

Strings in Python are immutable. This means their content cannot be changed after creation:

```
s = "Hello"
# s[0] = "h"  # This will raise an error
s = "hello"  # Instead, create a new string
print(s)
s1 = "Python"
s2 = s1
print(id(s1), id(s2))  # Same memory address (reference to the same object)
s1 = "Java"  # A new object is created
print(id(s1), id(s2))  # Different memory addresses
```

7 List

7.1 Creating Lists

A list can be created using square brackets `[]` or the `list()` constructor.

```
# Creating a list
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed = [1, "Python", 3.14]
print(numbers)  # Output: [1, 2, 3, 4, 5]
```

7.2 Slicing and Indexing

Lists support indexing to access individual elements and slicing to extract sublists.

```python
numbers = [10, 20, 30, 40, 50]
print(numbers[0])    # Output: 10
print(numbers[-1])   # Output: 50
print(numbers[1:4])  # Output: [20, 30, 40]
print(numbers[:3])   # Output: [10, 20, 30]
print(numbers[::2])  # Output: [10, 30, 50]
```

7.3 List Operations

Lists support operators for concatenation, repetition, and membership testing.

```python
list1 = [1, 2, 3]
list2 = [4, 5, 6]
print(list1 + list2)  # Output: [1, 2, 3, 4, 5, 6]
print(list1 * 2)      # Output: [1, 2, 3, 1, 2, 3]
print(2 in list1)     # Output: True
```

7.4 Built-in Functions

Python provides several built-in list functions:

```python
# append() - Adds an element to the end of the list
numbers = [1, 2, 3]
numbers.append(4)
print("append():", numbers)  # Output: [1, 2, 3, 4]

# count() - Counts occurrences of an element in the list
numbers = [1, 2, 2, 3, 4]
print("count():", numbers.count(2))  # Output: 2

# extend() - Extends the list by appending elements from another iterable
numbers = [1, 2, 3]
numbers.extend([4, 5])
print("extend():", numbers)  # Output: [1, 2, 3, 4, 5]

# index(elem, start, end) - Returns the index of the first occurrence of an element
numbers = [1, 2, 3, 4]
print("index():", numbers.index(3))  # Output: 2

# insert(pos, elem) - Inserts an element at a specific position
numbers = [1, 2, 4]
numbers.insert(2, 3)
print("insert():", numbers)  # Output: [1, 2, 3, 4]

# pop(pos) - Removes and returns an element at a specific index
numbers = [1, 2, 3, 4]
print("pop():", numbers.pop(2))  # Output: 3
print("After pop:", numbers)    # Output: [1, 2, 4]

# remove(elem) - Removes the first occurrence of an element
numbers = [1, 2, 3, 2]
numbers.remove(2)
print("remove():", numbers)  # Output: [1, 3, 2]

# reverse() - Reverses the list in place
numbers = [1, 2, 3]
numbers.reverse()
print("reverse():", numbers)  # Output: [3, 2, 1]

# sort(reverse=True|False, key=myFunc) - Sorts the list in ascending or descending order
numbers = [3, 1, 4, 2]
```

```
numbers.sort()
print("sort() ascending:", numbers)  # Output: [1, 2, 3, 4]
numbers.sort(reverse=True)
print("sort() descending:", numbers)  # Output: [4, 3, 2, 1]
```

7.5 Other Functions

```
# Python List Utility Functions Examples

# len() - Returns the number of elements in a list
numbers = [1, 2, 3, 4]
print("len():", len(numbers))  # Output: 4

# list() - Creates a list from an iterable
string = "Python"
print("list():", list(string))  # Output: ['P', 'y', 't', 'h', 'o', 'n']

# sum() - Returns the sum of elements in a list
numbers = [1, 2, 3, 4]
print("sum():", sum(numbers))  # Output: 10
```

7.6 Mutability

Lists are mutable, meaning their content can be changed. Assigning one list to another creates a reference.

```
mutable_list = [1, 2, 3]
mutable_list[0] = 100
mutable_list.append(4)
mutable_list.remove(2)
```

7.7 Reference

Assigning one list to another creates a reference. It means that when you assign one list to another in Python, both lists refer to the same list object in memory.

```
# Demonstrating list reference
original_list = [1, 2, 3]
reference_list = original_list
reference_list.append(4)

print(original_list)
print(reference_list)
print(original_list is reference_list)

>>>[1, 2, 3, 4]
>>>[1, 2, 3, 4]
>>>True
```

7.8 Shallow Copy

A shallow copy creates a new list but does not copy nested objects.

```
import copy
nested_list = [1, [2, 3], 4]
shallow_list = copy.copy(nested_list)
shallow_list[1].append(5)
```

```
shallow_list[0] = 100

print(nested_list)
print(shallow_list)
print(nested_list is shallow_list)

>>>[1, [2, 3, 5], 4]
>>>[100, [2, 3, 5], 4]
>>>False
```

7.9 Deep Copy

A deep copy creates a new list and recursively adds copies of the objects found in the original list.

```
# Demonstrating deep copy
import copy
nested_list = [1, [2, 3], 4]
deep_list = copy.deepcopy(nested_list)
deep_list[1].append(5)
print(nested_list)
print(deep_list)
print(nested_list is deep_list)

>>>[1, [2, 3], 4]
>>>[1, [2, 3, 5], 4]
>>>False
```

8 Tuple

8.1 Creating Tuples

A tuple is created using parentheses `()` with elements separated by commas.

```
# Creating a tuple with multiple elements
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)  # Output: (1, 2, 3, 4, 5)
single_element_tuple = (5,) #When the tuple contains only one element, you need to add a comma after the element,
                            #otherwise the parentheses will be used as operators.

list_to_tuple = tuple([1, 2, 3]) # Creating a tuple from a list
print(list_to_tuple)  # Output: (1, 2, 3)
```

8.2 Tuple Operations

```
# Concatenation
tuple1 = (1, 2)
tuple2 = (3, 4)
combined_tuple = tuple1 + tuple2
print(combined_tuple)  # Output: (1, 2, 3, 4)

# Repetition
repeated_tuple = ('A',) * 3
print(repeated_tuple)  # Output: ('A', 'A', 'A')

# Membership Testing
colors = ('red', 'blue', 'green')
print('red' in colors)  # Output: True
```

8.3 Tuple Built-in Functions

```
# Counts occurrences of an element.
nums = (1, 2, 3, 1, 1)
print(nums.count(1))  # Output: 3

# Finds the first occurrence of an element.
print(nums.index(3))  # Output: 2
```

8.4 Packing and Unpacking

```
# Packing a tuple
person = "Alice", 25, "Engineer"
print(type(person))

# Unpacking a tuple
name, age, profession = person
print(name)  # Output: Alice

#Using * to Capture Remaining Values
numbers = (1, 2, 3, 4, 5)
first, *middle, last = numbers
print(middle)  # Output: [2, 3, 4]


# A very useful application: Swap
a = 1
b = 2

a, b = b, a
print(a,b) # 2 1

```

8.5 Mutability

Tuple is **immutable**. Once a tuple is created, the elements in the tuple cannot be changed.

```
fruits = 'apple', 'banana', 'cherry'
fruits[1] = "watermelon"

Traceback (most recent call last):
  File "/Users/dengq/Documents/CSC111-2025Spring/project/main.py", line 2, in <module>
    fruits[1] = "watermelon"
TypeError: 'tuple' object does not support item assignment
```

8.6 Reference

A reference means that you have two names (variables) that refer to the exact same object in memory. Changes made through one reference are visible through the other. Since tuples are immutable, you can't really change them, but you can make one reference point to a different object. However, if a tuple contains mutable objects (like lists), the reference points to the same mutable objects.

```
tuple1 = (1, 2, [3, 4])
# tuple2 is now a reference to the same object as tuple1
tuple2 = tuple1

# Modifying the mutable element inside the tuple (the list)
```

```
tuple1[2].append(5)  # This changes the list inside the tuple

print(tuple1)
print(tuple2)
# They are the same object!
print(tuple1 is tuple2)
----------------------
(1, 2, [3, 4, 5])
(1, 2, [3, 4, 5])
True
```

8.7 Shallow Copy

A shallow copy creates a new object but fills it with references to the original objects found in the original. For tuples, this can be a bit tricky since tuples are immutable.

For immutable objects like tuples, a shallow copy doesn't really create a new distinct tuple if you use the copy.copy() method because tuples are immutable. When you attempt to shallow copy an immutable object like a tuple, Python may simply return a reference to the original object instead of creating a new object, since the immutable nature of tuples means that a separate copy is not needed for safety (as it cannot be changed anyway).

```
import copy

tuple1 = (1, 2, [3, 4])
tuple2 = copy.copy(tuple1)  # Shallow copy;For immutable objects like tuples,
                            #a shallow copy doesn't really create a new distinct tuple
print(tuple1 is tuple2)
print(tuple1 == tuple2)

# Modifying the mutable element inside the tuple (the list)
tuple1[2].append(5)

print(tuple1)
print(tuple2)
print(tuple1 is tuple2) # same object in system memory
print(tuple1 == tuple2) # equal value or not


----------------------
True
True
(1, 2, [3, 4, 5])
(1, 2, [3, 4, 5])
True
True
```

8.8 Deep Copy

A deep copy creates a new object and recursively adds copies of the objects found in the original. This means that not just the tuple itself but also the objects contained within it are copied.

For mutable objects inside the tuple, this ensures that changes to the mutable objects in the copied tuple do not affect the original tuple's contents.

```
import copy

tuple1 = (1, 2, [3, 4])
tuple2 = copy.deepcopy(tuple1)  # Deep copy; creates a new tuple and a new list inside it
# Modifying the mutable element inside the tuple (the list)

print(tuple1 is tuple2)
print(tuple1 == tuple2)

```

```
tuple1[2].append(5)
print(tuple1)
print(tuple2)

print(tuple1 is tuple2)
print(tuple1 == tuple2)
----------------------
False
True
(1, 2, [3, 4, 5])
(1, 2, [3, 4])
False
False
```

9 Set

A **set** in Python is an *unordered*, *mutable* collection of **unique** elements. Sets are excellent for operations that involve membership testing, removing duplicates, and efficiently computing mathematical operations such as union, intersection, and difference.

9.1 Creating Sets

The simplest way to create a set is by using curly braces {} with comma-separated elements:

```
# Creating a set of integers
my_set = {1, 2, 3, 4}
print(my_set)  # The order is not guaranteed

# You can also use the set() constructor to create a set from any iterable (e.g., lists, tuples, strings).
list_data = [1, 2, 2, 3, 4, 4]
set_from_list = set(list_data)
print(set_from_list)  # Duplicates are removed

set_from_string = set("hello")
print(set_from_string)  # {'h', 'e', 'l', 'o'}

# An empty set can only be created using the set() constructor:
empty_set = set()
print(empty_set)  # Output: set()
```

Using curly braces {} with nothing inside creates an empty *dictionary*, not a set.

9.2 Set Operations

```
A = {1, 2, 3}
B = {3, 4, 5}

# Union Operation
union_set = A | B
print(union_set)  # {1, 2, 3, 4, 5}
union_set = A.union(B)

# Intersection Operation
intersect_set = A & B
print(intersect_set)  # {3}
intersect_set = A.intersection(B)

# Difference Operation
diff_set = A - B
print(diff_set)  # {1,2}
```

```
diff_set = A.difference(B)

# Symmetric Difference
sym_diff = A ^ B
print(sym_diff)  # {1, 2, 4, 5}
sym_diff = A.symmetric_difference(B)

#Membership Check
fruits = {"apple", "banana", "cherry"}
print("apple" in fruits)   # True
print("orange" not in fruits)  # True
```

9.3 Set Built-in Functions

```
# Initial Sets
A = {1, 2, 3}
B = {3, 4, 5}

# add(): Adds a single element to the set A.
A.add(10)
print("After A.add(10):", A)

# Reset A for the next example
A = {1, 2, 3}

# update(): Adds multiple elements from B (or any iterable) to set A.
A.update(B)
print("After A.update(B):", A) # Result: A becomes {1, 2, 3, 4, 5}

# Reset A for the next example
A = {1, 2, 3}

# remove(): Removes an element from A. Raises KeyError if the element doesn't exist.
A.remove(2)
print("After A.remove(2):", A) # Result: A becomes {1, 3}
# If we tried A.remove(5) here, it would raise a KeyError since 5 isn't in A.

# Reset A for the next example
A = {1, 2, 3}

# discard(): Removes an element from A if present, else does nothing (no error).
A.discard(2)
print("After A.discard(2):", A) # Result: A becomes {1, 3}
A.discard(10)  # 10 is not in A, so nothing happens
print("After A.discard(10):", A) # Result: A is still {1, 3}

# Reset A for the next example
A = {1, 2, 3}

# pop(): Removes and returns an arbitrary (random) element from the set.
popped_value = A.pop()
print(f"After A.pop(): popped_value = {popped_value}, A =", A)
# Result: A loses one random element, popped_value holds that removed element
```

9.4 Hashability

In Python, an object is said to be **hashable** if it has a hash value that remains constant throughout its lifetime and it supports equality comparisons. When Python stores elements in a set or uses them as dictionary keys, it needs to determine whether two objects are the same in an efficient manner. This determination is done by comparing their hash values.

You can think of the hash function as a special function that takes an object and returns a number (often referred to as the "hash value"). If the object's data never changes, then the hash function will always return the same number. That is precisely what makes an object *hashable*.

On the other hand, if you could alter the object's contents, then its hash value would also need to change. This would break the expectation that a set or dictionary can rely on consistent hash values for fast lookups and equality checks. Which types are hashable?

- **Immutable types**, such as integers, strings, and tuples (contains immutable objects), are naturally hashable. They cannot change after creation, so their hash values remain stable.
- **Mutable types**, like lists or dictionaries, are not hashable because their contents can change. Such changes would alter the hash value and disrupt the internal consistency of sets and dictionaries.

9.5 Mutability

In Python, a set is considered a **mutable** data type, which means its contents can change over time. Specifically, you can:

- **Add elements** using `.add(x)` or `.update(iterable)`.
- **Remove elements** using `.remove(x)`, `.discard(x)`, or `.pop()`.

Because sets are mutable, they *cannot* be used as dictionary keys or be nested within other sets (those operations require hashability and, by extension, immutability). If you need an immutable, hashable set-like structure, Python provides the `frozenset` type.

9.6 Shallow Copy

A shallow copy of a set creates a new set object, but the elements within the new set are the same objects as the elements in the original set, because all elements are immutable.

```
import copy
set1 = {1, 2, 3}
set2 = copy.copy(set1) # Shallow copy

print(set1 is set2)
print(set1 == set2)

set1.add(4)  # Adding an element to set1

print(set1)
print(set2)

print(set1 is set2)
print(set1 == set2)

--------------------------
False
True
{1, 2, 3, 4}
{1, 2, 3}
False
False
```

9.7 Deep Copy

The deep copy performs the same as the shallow copy for the set, because the elements in a set are all immutable.

```
import copy
set1 = {1, 2, 3}
set2 = copy.deepcopy(set1) # Shallow copy

print(set1 is set2)
```

```
print(set1 == set2)

set1.add(4)  # Adding an element to set1

print(set1)
print(set2)

print(set1 is set2)
print(set1 == set2)

--------------------------
False
True
{1, 2, 3, 4}
{1, 2, 3}
False
False
```

10 Dictionary

A **dictionary** is a mutable, unordered collection in Python that stores data in *key-value* pairs. Each key is *unique* within the dictionary, and keys must be *hashable* (i.e., immutable types like strings, numbers, or tuples of immutables). Dictionaries are highly efficient for lookups, insertions, and deletions based on keys.

10.1 Creating Dictionaries

The most common way to define a dictionary is to use curly braces {} with pairs of `key:  value`.

```
# Creating an empty dictionary
empty_dict = {}

# Creating a dictionary with initial key-value pairs
person = {
    "name": "Alice",
    "age": 25,
    "location": "New York"
}
print(person)
# Possible output: {'name': 'Alice', 'age': 25, 'location': 'New York'}

You can also create dictionaries using the dict() constructor.
# Creating a dictionary from keyword arguments
person = dict(name="Bob", age=30, location="Paris")
print(person)
# Output: {'name': 'Bob', 'age': 30, 'location': 'Paris'}

# Creating a dictionary from a list of tuples
data_tuples = [("name", "Charlie"), ("age", 28)]
person = dict(data_tuples)
print(person)
# Output: {'name': 'Charlie', 'age': 28}
```

10.2 Accessing Dictionary Elements

Access the value of a dictionary by using the corresponding **key** in square brackets.

```
# Using key indexing
person = {"name": "Alice", "age": 25}
print(person["name"])  # Output: Alice

```

```
#To add a pair, you can direct use assignment
person["city"] = "NY"
print(person)
```

Attempting to access a nonexistent key raises a `KeyError`.

10.3 Dictionary Built-in Functions

```
# Define an initial dictionary
person = {"name": "Alice", "age": 25, "city": "New York"}

# get(key[, default])
#  - Returns the value for "key" if present, otherwise returns "default" if given, or None if not given.
print("   Name:", person.get("name"))              # "Alice"
print("   Country:", person.get("country"))        # None (because "country" is not a key)
print("   Country with default:", person.get("country", "Unknown"))  # "Unknown"

# items(): Returns a view of all key-value pairs as tuples.
print("   person.items():", person.items())
# Example output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])

# keys(): Returns a view of all keys in the dictionary.
print("   person.keys():", person.keys())
# Example output: dict_keys(['name', 'age', 'city'])

# values(): Returns a view of all values in the dictionary.
print("   person.values():", person.values())
# Example output: dict_values(['Alice', 25, 'New York'])

# update(other_dict): Updates the dictionary with key-value pairs from "other_dict".
person.update({"country": "USA", "age": 26})
print("   After update:", person)
# "country" is added, "age" is changed from 25 to 26.

# pop(key[, default]): Removes "key" and returns its value. If "key" is not found,
# returns "default" if given, otherwise raises KeyError.
removed_city = person.pop("city", "Not Found")
print("   Removed city:", removed_city)      # "New York"

# Attempting to pop a non-existing key:
removed_street = person.pop("street", "No street info")
print("   Removed street:", removed_street)  # "No street info"
```

You need to use `list()` function to turn dict_items, dict_keys, and dict_values into list data type for further usage in the program.

10.4 Mutability

A Python dictionary is a **mutable** data structure, meaning you can modify its contents after creation. Specifically:

- **Add or Update Key-Value Pairs:** You can add new keys by simple assignment or update existing values for existing keys.
- **Remove Items:** You can remove entries using methods like `pop`, or `popitem`.
- **Change Size Dynamically:** Unlike tuples or other immutable structures, a dictionary's size changes as you add or remove key-value pairs.

Because dictionaries are mutable and their contents can change, they cannot be used as dictionary keys or as elements of a `set`. For any key in a dictionary, Python requires its hash value to remain constant (hashable), which implies immutability.

10.5 Reference

Creating a reference to a dictionary means that you have two variables pointing to the same dictionary object in memory. Changes made through one reference are reflected in the other.

```
dict1 = {'a': 1, 'b': [2, 3, 4]}
dict2 = dict1  # dict2 is a reference to the same dictionary object as dict1
print(dict1 is dict2)
print(dict1 == dict2)

dict1['a'] = 5  # Changing value associated with 'a' in dict1
dict1['b'].append(5)  # Modifying the list inside the dictionary

print(dict1)
print(dict2)

print(dict1 is dict2)
print(dict1 == dict2)

----------------------
True
True
{'a': 5, 'b': [2, 3, 4, 5]}
{'a': 5, 'b': [2, 3, 4, 5]}
True
True
```

10.6 Shallow Copy

A shallow copy creates a new dictionary object, but inserts references to the objects found in the original dictionary. This means that changes to immutable values (like integers or strings) in the copied dictionary do not affect the original dictionary, but changes to mutable values (like lists) do.

```
import copy

dict1 = {'a': 1, 'b': [2, 3, 4]}
dict2 = copy.copy(dict1)  # dict2 is a reference to the same dictionary object as dict1
print(dict1 is dict2)
print(dict1 == dict2)

dict1['a'] = 5  # Changing value associated with 'a' in dict1
dict1['b'].append(5)  # Modifying the list inside the dictionary

print(dict1)
print(dict2)

print(dict1 is dict2)
print(dict1 == dict2)

-----------------------
False
True
{'a': 5, 'b': [2, 3, 4, 5]}
{'a': 1, 'b': [2, 3, 4, 5]}
False
False
```

10.7 Deep Copy

A deep copy creates a new dictionary object and recursively adds copies of the objects found in the original dictionary, ensuring that no objects are shared between the original and the copy.

```python
import copy

dict1 = {'a': 1, 'b': [2, 3, 4]}
dict2 = copy.deepcopy(dict1)  # dict2 is a reference to the same dictionary object as dict1
print(dict1 is dict2)
print(dict1 == dict2)

dict1['a'] = 5  # Changing value associated with 'a' in dict1
dict1['b'].append(5)  # Modifying the list inside the dictionary

print(dict1)
print(dict2)

print(dict1 is dict2)
print(dict1 == dict2)

-----------------------
False
True
{'a': 5, 'b': [2, 3, 4, 5]}
{'a': 1, 'b': [2, 3, 4]}
False
False
```

11 Function

A **function** in Python is a reusable block of code that performs a specific task. Functions improve code organization, reduce redundancy, and enhance modularity.

Basic Syntax:

```python
def function_name(parameters):
    """Optional docstring to describe the function."""
    # Function body
    # ...
    return value  # (optional)
```

Example: Simple Function

```python
def greet():
    print("Hello, welcome to Python functions!")

greet()  # Calling the function
```

11.1 Docstring

A docstring in Python is a special type of comment that appears as the first statement inside a function, class, or module. It is enclosed in triple quotes (""" or ''') and is used to document the functionality of the function.

```python
# A function docstring is placed immediately after the function definition.
def greet(name):
    """Return a greeting message for the given name."""
    return f"Hello, {name}!"
```

You can use **help()** to print it:

```
help(greet)

----------------------------
Help on function greet in module __main__:

greet(name)
    Return a greeting message for the given name.
```

11.2 Function Parameters and Arguments

11.2.1 Function Parameters

Definition: Parameters are the `placeholders` defined in the `function signature`. They act as variables inside the function and represent the values that the function will receive when called.

```
def greet(name): # name is a parameter for function greet
    print(f"Hello, {name}!")
```

11.2.2 Function Arguments

Definition: Arguments are the actual values that are passed to a function when calling it.

```
greet("Alice")  # "Alice" is an argument
Here, "Alice" is an argument because it is the actual value being passed to the function.
```

11.3 Positional Parameters

Positional parameters (also called positional arguments when used in a function call) are parameters that must be provided in the correct order when calling a function. The values passed to the function are assigned to the parameters based on their position.

```
def greet(name, major):
    print(f"Hello, my name is {name} and I study {major}.")

greet("Alice", "Computer Science")  # Correct order
```

If some parameters have default values, positional parameters must come first.

```
def introduce(name, country="USA"):
    print(f"My name is {name}, and I am from {country}.")

introduce("Bob")              # Output: My name is Bob, and I am from USA.
introduce("Alice", "Canada")  # Output: My name is Alice, and I am from Canada.
```

There will be an error if you define positional parameters that have default values in front of positional parameters.

```
def introduce(country="USA", name):
    print(f"My name is {name}, and I am from {country}.")

-----------------------------
File "./main.py", line 1
def introduce(country="USA", name):
                                ^
SyntaxError: non-default argument follows default argument
```

11.4 Keyword Parameters

Keyword arguments (`key=value`) explicitly specify which parameter gets the value.

```
def order(food, drink):
    print(f"I ordered {food} and {drink}.")

order("Pizza", "Coke")               # Positional arguments
order(food="Burger", drink="Juice")  # Keyword arguments
order("Sushi", drink="Tea")          # Mixing both
```

When mixing positional and keyword arguments, positional must come first:

```
def order(food, drink):
    print(f"I ordered {food} and {drink}.")

order(food="Burger", "Juice")
----------------------------
File "./main.py", line 4
order(food="Burger", "Juice")
                           ^
SyntaxError: positional argument follows keyword argument
```

11.5 Positional-Only and Keyword-Only Parameters

In Python 3.8+, you can use both / (positional-only) and * (keyword-only) in the same function to control how arguments must be passed. The '/' ensures previous parameters are positional-only parameters, while '*' forces the rest to be keyword-only.

```
def student_info(name, major, /, *, university, year):
    print(f"Name: {name}, Major: {major}, University: {university}, Year: {year}")

# Correct usage
student_info("Alice", "CS", university="Harvard", year=2025)

# Incorrect usage (Positional required for `name` and `major`)
student_info(name="Alice", major="CS", university="Harvard", year=2025)  # TypeError

# Incorrect usage (Keyword required for `university` and `year`)
student_info("Alice", "CS", "Harvard", 2025)  # TypeError
```

- Before '/' → 'name' and 'major' must be positional.
- After '*' → 'university' and 'year' must be keyword arguments.

We can mix regular, positional-only, and keyword-only parameters for fine control. The parameters between '/' and '*' can be either positional or keyword parameters.

```
def describe_course(course_name, instructor, /, credits, *, semester):
    print(f"Course: {course_name}, Instructor: {instructor}, Credits: {credits}, Semester: {semester}")

# Correct usage
describe_course("Data Structures", "Dr. Smith", 3, semester="Fall 2024")
describe_course("Data Structures", "Dr. Smith", credits=3, semester="Fall 2024")
```

Parameter Type	Syntax Example	Behavior
Positional-only	'def func(a, b, /):'	'a' and 'b' must be passed by position.
Keyword-only	'def func(*, x, y):'	'x' and 'y' must be passed by keyword.
Mixed ('/' and '*')	'def func(a, /, b, *, c):'	'a' is positional-only, 'b' can be either, 'c' is keyword-only.

- **Use '/' (Positional-Only) When:** The parameter name is not important for users. Example: Built-in functions like `math.pow(x, y, /)`.
- **Use '*' (Keyword-Only) When:** You want clarity in function calls. Example: `send_email(subject, *, recipient, body)`

11.6 Variable-Length Parameter (*args and **kwargs)

11.6.1 Using *args (Multiple Positional Arguments):

The *args parameter collects all extra positional arguments into a **tuple**.

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

print(sum_all(1, 2, 3, 4, 5))  # Output: 15
```

You can specify positional-only parameters before /, and allow variable-length positional arguments (*args) after.

```
def process_numbers(a, b, /, *args):
    print(f"First: {a}, Second: {b}, Additional: {args}")

# Correct usage
process_numbers(1, 2, 3, 4, 5)  # Output: First: 1, Second: 2, Additional: (3, 4, 5)

# Incorrect usage (Cannot use keyword arguments for `a` and `b`)
process_numbers(a=1, b=2, 3, 4, 5)  # TypeError
```

11.6.2 Using **kwargs (Multiple Keyword Arguments):

The **kwargs parameter collects all extra keyword arguments into a **dictionary**.

```
def display_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=30, city="New York")
```

The * or *args force the remaining parameters to be keyword-only, and **kwargs collects additional keyword arguments.

```
def describe_person(name, /, *, age, **kwargs):
    print(f"Name: {name}, Age: {age}, Other: {kwargs}")

# Correct usage
describe_person("Alice", age=25, city="New York", job="Engineer")
# Output: Name: Alice, Age: 25, Other: {'city': 'New York', 'job': 'Engineer'}

# Incorrect usage (Keyword required for `age`)
describe_person("Alice", 25, city="New York")  # TypeError

def complex_function(x, y, /, *args, key1, key2, **kwargs):
    print(f"x: {x}, y: {y}")
    print(f"Additional Positional: {args}")
    print(f"Key1: {key1}, Key2: {key2}")
    print(f"Other Keyword Arguments: {kwargs}")

# Correct usage
complex_function(1, 2, 3, 4, 5, key1="A", key2="B", extra1="C", extra2="D")
```

11.7 Return Statement

In Python, a function can return zero (no return statement is needed), one or multiple values using the return statement.

Returning one value:

```
def square(num):
    return num * num

result = square(4)
print(result)  # Output: 16
```

In Python, a function can return multiple values by separating them with commas. These values are automatically packed into a tuple.

```
def get_student_info():
    name = "Alice"
    age = 22
    major = "Computer Science"
    return name, age, major  # Returns a tuple

info = get_student_info()
print(info)        # Output: ('Alice', 22, 'Computer Science')
print(info[0])     # Output: Alice
print(info[1])     # Output: 22
print(info[2])     # Output: Computer Science

name, age, major = get_student_info() # or directly unpacking
```

If a function has no return statement, it automatically returns **None**.

```
def say_hello():
    print("Hello!")

result = say_hello()
print(result)  # Output: None
```

11.8 Lambda Functions

A lambda function in Python is a small, anonymous function that can have any number of arguments but only one expression. It is often used for short, simple operations where defining a full function is unnecessary.

The basic syntax of a lambda function is:

```
lambda arguments: expression
#lambda: The keyword for defining a lambda function.
#arguments: One or more inputs, similar to parameters in a normal function.
#expression: A single expression that is evaluated and returned.

```

A simple example:

```
square = lambda x: x * x
print(square(5))  # Output: 25

add = lambda a, b: a + b
print(add(3, 7))  # Output: 10
```

Lambda functions are useful for sorting lists of tuples or dictionaries.

```
students = [("Alice", 25), ("Bob", 20), ("Charlie", 22)]
sorted_students = sorted(students, key=lambda student: student[1])  # Sort by age
print(sorted_students)
# Output: [('Bob', 20), ('Charlie', 22), ('Alice', 25)]

students = [{"name": "Alice", "age": 25}, {"name": "Bob", "age": 20}, {"name": "Charlie", "age": 22}]
sorted_students = sorted(students, key=lambda student: student["age"])
print(sorted_students)
```

11.9 Recursive Functions

A recursive function consists of:

- **Base Case**: A condition to stop recursion.
- **Recursive Case**: The function calls itself with a smaller input.

```
def recursive_function(parameters):
    if base_case_condition:
        return base_case_value  # Stops recursion
    return recursive_function(smaller_problem)  # Calls itself
```

Example: Factorial Using Recursion

```
def factorial(n):
    if n == 0 or n == 1:  # Base Case
        return 1
    return n * factorial(n - 1)  # Recursive Case

print(factorial(5))  # Output: 120
```

How it Works?

```
factorial(5) → 5 * factorial(4)
factorial(4) → 4 * factorial(3)
factorial(3) → 3 * factorial(2)
factorial(2) → 2 * factorial(1)
factorial(1) → 1  (Base Case)

Then, values are multiplied on the way back:

factorial(1) <- 1
factorial(2) <- 2 * 1 = 2
factorial(3) <- 3 * 2 = 6
factorial(4) <- 4 * 6 = 24
factorial(5) <- 5 * 24 = 120
```

Let's do some exercises:

- Sum of Natural Numbers using Recursion
- Reverse a String using Recursion

11.10 Higher-Order Functions

A **higher-order function** is a function that takes another function as an argument or returns a function. This allows for powerful functional programming techniques.

For this course, we will only discuss three built-in higher-order functions.

map():

The map() function applies a function to every item in an iterable and returns a map object.

```
nums = [1, 2, 3, 4]
squared_nums = list(map(lambda x: x ** 2, nums))
print(squared_nums)  # Output: [1, 4, 9, 16]
```

filter():

The filter() function filters elements based on a boolean function.

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

reduce():

The reduce() function (from functools) applies a function cumulatively to reduce a list to a single value.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 120
```

12 Conditional Statement

Conditional statements in Python allow the program to execute specific blocks of code based on Boolean conditions. These conditions evaluate to either `True` or `False`, making them essential for decision-making in programming.

12.1 Boolean Context

In Python, conditional statements rely on values that evaluate to `True` or `False`. Python treats some values as **truthy** (evaluating to `True`) and others as **falsy** (evaluating to `False`). The Boolean type has two possible values:

- `False` (equal to 0)
- `True` (equal to 1)

The following values evaluate to `False` in a Boolean context:

- Numbers: `0`, `0.0`
- Strings: `""` (empty string)
- Containers: `[]` (empty list), `()` (empty tuple), {} (empty dictionary), `set()` (empty set)
- `None` (the absence of a value)

Any non-zero number, non-empty string, or container with at least one element evaluates to `True`.

12.2 The `if` Statement

The `if` statement executes a block of code only if a condition evaluates to `True`.

```
#Syntax
if condition:
    # Code block executed if condition is True

#Example
x = 10
if x > 5:
    print("x is greater than 5")
```

12.3 The `if-else` Statement

The `if-else` statement provides an alternative block of code to execute if the condition is `False`.

```
#Syntax
if condition:
    # Code executed if condition is True
else:
    # Code executed if condition is False

#Example
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is NOT greater than 5")
```

12.4 The `if-elif-else` Statement

The `if-elif-else` statement checks multiple conditions sequentially. You can have multiple `elif` statements.

```
#Syntax
if condition1:
    # Code executed if condition1 is True
elif condition2:
```

```
    # Code executed if condition1 is False and condition2 is True
else:
    # Code executed if all conditions are False

#Example
x = 15

if x < 10:
    print("x is less than 10")
elif x < 20:
    print("x is between 10 and 20")
else:
    print("x is 20 or greater")
```

12.5 The Ternary Conditional Operator

Python supports a shorthand `if-else` expression known as the ternary conditional operator.

```
#Syntax

value_if_true if condition else value_if_false

#Example

x = 5
result = "Positive" if x > 0 else "Negative"
```

12.6 Conpound Conditions

Python provides logical operators:

- `and` (both conditions must be `True`)
- `or` (at least one condition must be `True`)
- `not` (negates a condition)

```
x = 8
if x > 5 and x < 10:
    print("x is between 5 and 10")
```

12.7 Short-Circuit Evaluation in Python

Python short-circuits logical evaluations to improve efficiency: - For `and`, if the first condition is `False`, the rest are **not evaluated**. - For `or`, if the first condition is `True`, the rest are **not evaluated**.

```
x = 10
if x > 5 or x / 0 == 1:
    print("Only the first condition is checked")
```

12.8 Exercises

12.8.1 Exercise 1

Write a function:

1. accept a grade as an argument
2. if grade is greater than 90, print “A”

3. if grade is between 80 and 90, print "B"
4. if grade is between 70 and 80, print "C"
5. if grade is between 60 and 70, print "D"
6. if grade is below 60, print "F"

Can you simplify your code based on the fact that the conditions are checked in a strict order?

12.8.2 Exercise 2

Write a function to determine whether the given year is a leap year or not.

1. accept a year as an argument
2. If a year is a leap year, print "[the input year] is a leap year.
3. If a year is not a leap year, print "[the input year] is a leap year.
4. for year __00 (divisible by 100) is a leap year if the year is divisible by 400
5. for the other year which is a leap year if the year is divisible by 4.

This is a very classic binary output question; now think about how to use conpound conditions to solve it use a simple `if-else` statement.

12.8.3 Exercise 3

Write a function:

1. accept three numbers as arguments
2. print the middle one
3. you cannot use any built-in functions

Unlike the previous leap year question, for this one, you need to be careful to give a conprehensive consideration, how many situations when a number is the middle of three?

12.8.4 Exercise 4

Write a function to check the input letter is vowel or not:

1. accept a single letter as an argument
2. if more than one, return error message
3. if it is vowel, print "it is vowel"
4. if not, print "it is not vowel."

Once again, this is a binary question—a letter is either a vowel or not. However, rather than comparing it against every possible vowel, is there a simpler way to solve this using a basic `if-else` statement?

13 Loops

Loops in Python allow us to execute a block of code multiple times efficiently.

13.1 The `for` Loop

The `for` loop iterates over a sequence (list, tuple, string, dictionary, or range).

```
#Syntax
for variable in sequence:
    # Code to execute in each iteration

#Example
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 by default, and stops before a specified number. `range(start, stop, step)`, remember, we have inclusive starting boundary and exclusive ending boundary `[start, stop)`.

```
for number in range(1, 6):
   print(number)

for number in range(1, 6, 2):
   print(number)

for number in range(6, 1, -1):
   print(number)
```

Nested for loop means a loop can be inside another loop.

```
for i in range(1,6):
  for j in range(1, i+1):
     print("*",end='')
  print('\r')
```

Loop control statements modify the behavior of loops. `break` stops the current loop immediately when encountered.

```
for i in range(10):
    if i == 5:
        break
    print(i)

for i in range(1,6):
  for j in range(1, 6):
     if i == 3:
        break
     print(i,j)
```

`continue` skips the current iteration and moves to the next one.

```
for i in range(10):
    if i == 5:
        continue
    print(i)

```

```
for i in range(1,6):
  for j in range(1, 6):
     if i == 3:
        continue
     print(i,j)
```

`pass` does nothing and acts as a placeholder.

```
for i in range(3):
    pass  # Placeholder for future code
```

`for ... else`: when the loop is executed (that is, after traversing all the elements in the iterable), the code in the else clause will be executed. If a break statement is encountered during the loop, the loop will be interrupted, and the else clause will not be executed at this time.

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")

for x in range(6):
    if x == 3:
        break
else:
    print("Finally finished!")
```

There are some useful functions:

```
techs = ["Apple", "Google","Meta","Microsoft"]
for index, tech in enumerate(techs):
   print(index, tech)

#zip in for loop
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
cities = ['New York', 'Los Angeles', 'Chicago']

# Combine all three lists
for name, age, city in zip(names, ages, cities):
   print("{}, {}, lives in {}.".format(name, age, city))
```

Exercise: print a multiplication chart:

```
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12  4x4=16
1x5=5 2x5=10  3x5=15  4x5=20  5x5=25
1x6=6 2x6=12  3x6=18  4x6=24  5x6=30  6x6=36
1x7=7 2x7=14  3x7=21  4x7=28  5x7=35  6x7=42  7x7=49
1x8=8 2x8=16  3x8=24  4x8=32  5x8=40  6x8=48  7x8=56  8x8=64
1x9=9 2x9=18  3x9=27  4x9=36  5x9=45  6x9=54  7x9=63  8x9=72  9x9=81
```

13.2 The `while` Loop

The `while` loop runs as long as the given condition is `True`.

```
#Syntax
while condition:
    # Code block executed as long as condition is True

#Example
i = 1
while i <= 5:
    print(i)
    i += 1
```

Infinite loop:

```
while True :  # infinite loop
 num = input("enter a number  :")
 if num == 'exit':
   break;
 print ("the number you entered is: ", int(num))

print ("Good bye!")
```

`while ... else`: when the loop is executed (that is, after condition becomes false), the code in the else clause will be executed. If a break statement is encountered during the loop, the loop will be interrupted, and the else clause will not be executed at this time.

```
numbers = [1, 3, 5, 7, 9]
target = 4 # change it to a number in the above list to see the difference
index = 0

while index < len(numbers):
    if numbers[index] == target:
        print(f"Found {target} at index {index}")
        break  # Exit loop if the element is found
    index += 1
else:
    print(f"{target} was not found in the list")  # Executes only if loop runs completely

```

`break`, `continue`, and `pass` statements function similarly in a `while` loop as they do in a for loop.

Exercise: Write a Python function to let the user guess an integer number between 1 and 9.

1. randomly generate an integer between 1 and 9
2. The user is prompted to enter a guess between 1 and 9.
3. If the user guesses wrong, then print "try again!" and the prompt appears again to let the user enter a guess until the guess is correct.
4. On successful guess, print the "Well guessed!" message, and the program will exit.

13.3 Comprehensions

Python comprehensions allow concise and efficient creation of collections.

```
#Creates a list using a single-line loop.
squares = [x**2 for x in range(1, 6)]
print(squares)

#List comprehension with condition
even_numbers = [x for x in range(10) if x % 2 == 0]
print(even_numbers)

```

```python
#Dictionary comprehension
squares_dict = {x: x**2 for x in range(1, 6)}
print(squares_dict)

#Set comprehension
unique_squares = {x**2 for x in [1, 2, 2, 3, 4, 4]}
print(unique_squares)
```

There are two functions that are often used with conprehensions.

```python
# all() to check if all elements are true
numbers = [10, 7, 6, 9, 8]
result = all(num > 5 for num in numbers)
print("Are all numbers greater than 5?", result)

# any() to check if any element is true
numbers = [5, 7, 12, 3, 8]
result = any(num > 10 for num in numbers)
print("Is there any number greater than 10?", result)
```

13.4 What is Iterable?

An iterable is any Python object capable of returning its members one at a time, allowing it to be used in a `for` loop, list comprehension, generator expression, or other iteration contexts.

Iterables in Python have several key characteristics:

- The most common use of iterables is in `for` loops:

```python
for item in iterable:
# Process item
```

- You can check if an element exists in an iterable:

```python
if element in iterable:
# Do something
```

Technically, an iterable must implement the `__iter__()` method that returns an iterator, or it must implement the `__getitem__()` method that can take sequential indexes (starting from 0). We will talk later in the Objected Oriented Programming chapter.

13.4.1 Common Built-in Iterables

Python has many built-in types that are iterables:

- **Lists**: `[1, 2, 3]`
- **Tuples**: `(1, 2, 3)`
- **Strings**: `"hello"` (iterates through characters)
- **Dictionaries**: `"a": 1, "b": 2` (iterates through keys by default)
- **Sets**: `1, 2, 3`
- **Files**: iterates through lines
- **Range objects**: `range(5)`
- **Generators**

13.4.2 Iterables vs. Iterators

It's important to understand the distinction between iterables and iterators:

- An **iterable** is an object that has an `__iter__()` method which returns an iterator, or which defines a `__getitem__()` method that can take sequential indexes.
- An **iterator** is an object that implements the iterator protocol, which consists of the methods `__iter__()` and `__next__()`.

When you use a `for` loop on an iterable, Python automatically creates an iterator from the iterable and then calls `__next__()` repeatedly until a `StopIteration` exception is raised.

```
i = iter([1,2,3])
print(next(i))
print(next(i))
print(next(i))
print(next(i))

------------------------
1
2
3
Traceback (most recent call last):
  File "project/main.py", line 7, in <module>
    print(next(i))
StopIteration
```

13.4.3 Checking if an Object is Iterable

You can check if an object is iterable using the `isinstance()` function:

```
from collections.abc import Iterable
obj = [1,2,3]
print(isinstance(obj, Iterable))
```

13.5 What Is A Generator?

Generators are a special type of iterator that allow you to iterate over a potentially large sequence of data without loading the entire sequence into memory at once. They generate values on-the-fly during iteration.

The simplest way to create a generator is using a function with the `yield` keyword:

```
def simple_generator():
    yield 1
    yield 2
    yield 3

#Using the generator
gen = simple_generator()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
print(next(gen))  # StopIteration exception
```

Every time `yield` is called, the function's state is "frozen," and the yielded value is returned. When `next()` is called again, execution resumes from where it left off.

Similar to list comprehensions, but using parentheses instead of square brackets:

```python
#squares_list uses much more memory than squares_gen

#List comprehension (creates entire list in memory)
squares_list = [x*x for x in range(1000000)]
#Generator expression (creates values on-demand)
squares_gen = (x*x for x in range(1000000))
```

Generators produce values one at a time, which is ideal for large datasets.

```python
#Generator Example: Fibonacci Sequence Generator
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

#Generate first 10 Fibonacci numbers
fib = fibonacci()
for _ in range(10):
    print(next(fib), end=" ")
#Output: 0 1 1 2 3 5 8 13 21 34
```

1. **Generators are one-time use**: Once exhausted, you need to recreate them.
2. **No random access**: You can only iterate sequentially.
3. **Generator expressions in function calls**: When used as the only argument, parentheses can be omitted.

```python
sum(x*x for x in range(10))  # No need for extra parentheses
```

14 Introduction to File Operations in Python

Python provides a rich set of tools and methods for working with files, making it easy to read, write, and manipulate data in various formats.

14.1 Opening and Closing Files

The most fundamental file operation is opening a file. In Python, the `open()` function is used to open files.

```python
#Basic syntax for opening a file
file = open('test.txt', 'mode')
#Process the file...
#Always close the file when done
file.close()
```

The `mode` parameter specifies how the file should be opened:

- `'r'`: Read mode (default)
- `'w'`: Write mode (creates a new file or truncates an existing file)
- `'a'`: Append mode (adds content to the end of the file)
- `'x'`: Exclusive creation mode (fails if the file already exists)
- `'b'`: Binary mode (used with other modes, e.g., `'rb'` for reading binary)
- `'t'`: Text mode (default, used with other modes)
- `'+'`: Update mode (allows both reading and writing)

The preferred way to work with files in Python is using the `with` statement, which automatically handles proper closing of files, even when exceptions occur:

```
#Using context manager to open a file
with open('test.txt', 'r') as file:
    content = file.read()
    # Process content...
    #File is automatically closed when the block ends
```

14.2 Reading from Files

Python offers several methods to read file content:

```
#Reading the entire file as a string
with open('test.txt', 'r') as file:
    content = file.read()
    print(content)

#Reading one line
with open('test.txt', 'r') as file:
    line = file.readline()  # Reads a single line
    print(line)

#Reading all lines into a list
with open('test.txt', 'r') as file:
    lines = file.readlines()  # Returns a list of lines
    for line in lines:
        print(line.strip())  # strip() removes trailing newline
        #strip can remove whitespace including space, \t, \n, \r, etc

#Iterating over the file object (memory efficient)
with open('test.txt', 'r') as file:
    for line in file:  # File objects are iterable
        print(line.strip())
```

14.3 Writing to Files

Writing to files is just as straightforward:

```
#Writing a string to a file (overwrites existing content)
with open('test.txt', 'w') as file:
    file.write('Hello, World!\n')
    file.write('This is a new line.')

#Writing multiple lines at once
lines = ['First line', 'Second line', 'Third line']
with open('test.txt', 'w') as file:
    file.writelines(line + '\n' for line in lines)

#Appending to a file
with open('test.txt', 'a') as file:
    file.write('New log entry\n')
```

14.4 Working with File Paths

The `os.path` module provides functions for working with file paths in a cross-platform manner:

```python
import os.path
#Joining path components
path = os.path.join('folder', 'subfolder', 'file.txt')
print(path)  # 'folder/subfolder/file.txt' (Unix) or 'folder\subfolder\file.txt' (Windows)

#Getting the absolute path
abs_path = os.path.abspath('test.txt')
print(abs_path)

#Checking if a path exists
exists = os.path.exists('test.txt')
print(f"File exists: {exists}")

#Checking if a path is a file or directory
is_file = os.path.isfile('test.txt')
is_dir = os.path.isdir('folder')
print(is_file, is_dir)

#Splitting paths
dirname, filename = os.path.split('/path/to/file.txt')
print(f"Directory: {dirname}, Filename: {filename}")

#Getting file extension
filename, extension = os.path.splitext('test.txt')
print(f"Filename: {filename}, Extension: {extension}")
```

14.5 Creating and Removing Directories

Python provides functions for managing directories:

```python
import os
import shutil

#Create a directory
os.mkdir('new_folder')

#Create nested directories
os.makedirs('parent/child/grandchild', exist_ok=True)

#Remove an empty directory
os.rmdir('empty_folder')

#Remove a directory and its contents
shutil.rmtree('folder_with_contents')

#List all files and directories
entries = os.listdir('folder') # return a list
for entry in entries:
    print(entry)

#List with full paths
with os.scandir('folder') as entries:
    for entry in entries:
        print(entry.name)
        print(f"Is file: {entry.is_file()}")
        print(f"Is directory: {entry.is_dir()}")
        print(f"Path: {entry.path}")

#Walking a directory tree
for root, dirs, files in os.walk('folder'):
    print(f"Current directory: {root}")
    print(f"Subdirectories: {dirs}")
    print(f"Files: {files}")
```

```

#Moving/renaming files
os.rename('old_name.txt', 'new_name.txt')
shutil.move('file.txt', 'new_location/file.txt')

#Removing files
os.remove('file_to_delete.txt')
```

14.6 CSV Files

The `csv` module simplifies working with CSV (Comma-Separated Values) files:

```
import csv

with open('test.csv', 'w') as csvfile:
    fieldnames = ['Name', 'Age', 'City']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'Name': 'Eve', 'Age': 22, 'City': 'Miami'})
    writer.writerow({'Name': 'Frank', 'Age': 40, 'City': 'Seattle'})

with open('test.csv', 'r') as csvfile:
    dict_reader = csv.DictReader(csvfile)
    for row in dict_reader:
        print(row)  # row is a dictionary
```

15 Errors and Exceptions

Python provides a comprehensive exception handling mechanism that allows programmers to gracefully handle errors and take appropriate actions when they occur.

15.1 Errors vs. Exceptions

In Python, there's a technical distinction between errors and exceptions:

- **Errors** typically refer to serious problems that a reasonable application should not try to catch. Examples include syntax errors (forget to put a : after `else`).
- **Exceptions** are events that occur during program execution that disrupt the normal flow of instructions. These can be caught and handled by the application (divided by zero).

15.2 Types of Errors in Python

Python errors generally fall into three categories:

1. **Syntax Errors**: Errors in the structure of your Python code that prevent it from being parsed properly. Syntax errors must be fixed before your code can run.
2. **Runtime Errors** (Exceptions): Errors that occur during program execution.

```
#ZeroDivisionError
result = 10 / 0

#TypeError
text = "hello" + 5

#NameError
print(undefined_variable)

#IndexError
my_list = [1, 2, 3]
print(my_list[10])
```

```
13
14  #KeyError
15  my_dict = {"name": "John"}
16  print(my_dict["age"])
17
18  #FileNotFoundError
19  with open("nonexistent_file.txt", "r") as file:
20  content = file.read()
```

3. **Logical Errors**: Errors in the logic of your program that cause it to behave incorrectly but don't necessarily cause it to crash. Logical errors are the most challenging to find because they don't produce error messages. The program runs without crashing, but produces incorrect results due to flaws in its logic.

15.3 The try-except Block

The basic structure for handling exceptions in Python is the `try-except` block:

```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code to handle the exception
    print("Error: Division by zero!")
```

When an exception occurs in the `try` block, Python looks for a matching `except` block to handle it. If found, execution continues in that block; otherwise, the exception propagates up the call stack.

You can handle different exceptions with separate `except` blocks:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Please enter a valid number!")
```

You can also handle multiple exception types with a single `except` block by specifying them as a tuple:

```
try:
    # Some code that might raise exceptions
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError):
    print("Error: Invalid input or division by zero!")
```

You can catch all exceptions with a generic `except` clause:

```
try:
    # Some code that might raise exceptions
    num = int(input("Enter a number: "))
    result = 10 / num
except:  # Catches all exceptions
    print("An error occurred!")
```

However, this approach is generally discouraged because:

- It catches *all* exceptions, including those you might not want to catch (like `KeyboardInterrupt`).

- It makes debugging more difficult because you don't know what specific exception occurred.

A better approach is to catch `Exception`, which covers most exceptions but not system exits, keyboard interrupts, etc.:

```
try:
    # Some code that might raise exceptions
    num = int(input("Enter a number: "))
    result = 10 / num
except Exception:
    print("An error occurred")
```

15.4 Accessing Exception Information

You can access information about the exception using the `as` keyword:

```
try:
    x = 1 / 0
except ZeroDivisionError as err:
    print(f"Error details: {err}")
    print(f"Error type: {type(err).__name__}")
```

This gives you access to the exception instance, which often contains useful information about what went wrong.

15.5 The else Clause

You can use the `else` clause to run code when no exceptions occur:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
else:
    # This block runs if no exceptions were raised
    print(f"Result: {result}")
```

This pattern is useful for separating the exception-generating code from the code that should run when no errors occur.

15.6 The finally Clause

The `finally` clause contains code that always executes, regardless of whether an exception occurred:

```
try:
    file = open("data.txt", "r")
    content = file.read()
    # Process content...
except FileNotFoundError:
    print("Error: File not found!")
finally:
    # This block always executes
    # It's used for cleanup operations
    if 'file' in locals() and not file.closed:
        file.close()
    print("File closed.")
```

The `finally` block is ideal for **cleanup operations**, like closing files or network connections, that should happen whether or not an exception occurred.

You can use both `else` and `finally` in the same `try` block:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

15.7 Raising Exceptions

You can manually raise exceptions using the `raise` statement:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age
try:
    user_age = set_age(-5)
except ValueError as e:
    print(f"Error: {e}")
```

You can also raise a different exception after catching one:

```
try:
    file = open("config.txt", "r")
except FileNotFoundError:
    # Convert the exception to a more specific one
    raise RuntimeError("Configuration file is missing")
```

Or you can modify and re-raise the same exception:

```
try:
    num = int("abc")
except ValueError as e:
    # Add context to the exception
    raise ValueError(f"Invalid input") from e
```

The `from e` syntax creates an **exception chain**, which helps maintain the original exception information.

Why we use "try ... except ... else ... finally" instead of "if ... else":

Because if ... else... will always be executed! If the condition is 99% true, you should use try except. if the condition is 50% false, well, if .. else is better, because for a single call, try except cost more time and resources than if .. else.

15.8 An Example

```
def read_config():

    try:
        # Opening a resource (file in this case)
        with open("config.txt", "r") as file:
            config = file.read()
```

```
            print(f"Read configuration: {config}")

            # Inner try block for processing the data
            try:
                # Convert the configuration to an integer
                value = int(config.strip())
                result = 100 / value  # This may cause ZeroDivisionError
                print(f"Calculation result: {result}")

            except ValueError:
                print("Inner exception: Configuration is not a valid number")
                # Handle this specific error but continue execution
                result = 0

            except ZeroDivisionError:
                print("Inner exception: Cannot divide by zero")
                # Handle this specific error but continue execution
                result = float('inf')

            # Continue with the rest of the processing
            print(f"Proceeding with result: {result}")

    except FileNotFoundError:
        print("Outer exception: Configuration file not found")
        # Create default configuration
        with open("config.txt", "w") as file:
            file.write("10")
        print("Created default configuration file")

    finally:
        print("Cleaning up resources...")
        # This block always executes, regardless of whether exceptions occurred

    print("Operation completed")
```

15.9 What are Assertions?

Assertions in Python are a debugging aid that test a condition as a boolean expression. If the condition is True, the program continues execution as normal. If the condition is False, the program raises an `AssertionError` exception, which can optionally include an error message.

The primary purpose of assertions is to catch bugs and logical errors during development and testing. They're designed to verify that certain conditions or assumptions in your code are indeed true at specific points in the program's execution.

The basic syntax of the Python `assert` statement is:

```
assert condition[, error_message]
```

Where:

- `condition` is an expression that is expected to evaluate to True
- `error_message` is an optional string that will be included in the exception if the assertion fails

Functionally, an assert statement is roughly equivalent to:

```
if not condition:
    raise AssertionError(error_message)
```

Let's start with a simple example that demonstrates how assertions work:

```
def calculate_average(numbers):
    assert len(numbers) > 0, "Cannot calculate average of empty list"
    return sum(numbers) / len(numbers)

try:
    print(calculate_average([]))
except AssertionError as e:
    print(f"Caught an error: {e}")  # Output: Caught an error: Cannot calculate average of empty list
```

In this example, the assertion verifies that the input list contains at least one element before attempting to calculate the average. This prevents a division by zero error and makes the function's precondition explicit.

15.10 When and How to Use Assertions

Assertions are most appropriate for the following scenarios:

1. **Validating preconditions**: Checking that the inputs to a function satisfy necessary conditions.

```
def calculate_square_root(number):
    # Precondition: number must be non-negative
    assert number >= 0, f"Cannot calculate square root of negative number: {number}"

    return number ** 0.5

# This works
print(calculate_square_root(16))  # Output: 4.0

# This would fail the assertion
try:
    print(calculate_square_root(-4))
except AssertionError as e:
    print(e)  # Output: Cannot calculate square root of negative number: -4
```

2. **Verifying postconditions**: Ensuring that a function's output matches expected properties

```
def sort_numbers(numbers):
    # Function to sort a list of numbers in ascending order
    result = sorted(numbers)

    # Postcondition: result should be sorted
    assert all(result[i] <= result[i+1] for i in range(len(result)-1)), "Result is not properly sorted"

    # Postcondition: result should contain the same elements as the input
    assert len(result) == len(numbers), "Elements were lost or added during sorting"

    return result

# This works
print(sort_numbers([3, 1, 4, 1, 5, 9]))  # Output: [1, 1, 3, 4, 5, 9]
```

3. **Checking invariants**: Verifying that certain properties remain true throughout execution

```
def find_in_sorted_list(sorted_list, target):
    # Use binary search to find target in a sorted list
    left, right = 0, len(sorted_list)

    while left < right:
        # Invariant: if target exists in the list, it must be in the range sorted_list[left:right+1]
        assert left >= 0 and right <= len(sorted_list), f"Search indices out of bounds: left={left}, right={right}"

```

```
        mid = (left + right) // 2

        if sorted_list[mid] == target:
            return mid
        elif sorted_list[mid] < target:
            left = mid + 1
        else:
            right = mid

    return -1  # Target not found

# This works
sorted_data = [1, 3, 5, 7, 9, 11, 13, 15]
print(find_in_sorted_list(sorted_data, 7))  # Output: 3 (index of 7 in the list)
print(find_in_sorted_list(sorted_data, 6))  # Output: -1 (not found)
```

4. **Documenting assumptions**: Making implicit assumptions explicit in the code

```
def process_user_data(user_dict):
    # Documenting assumption: user_dict contains a 'role' key
    assert 'role' in user_dict, "User data must contain a 'role' field"

    # Documenting assumption: valid roles are 'admin', 'user', or 'guest'
    assert user_dict['role'] in ['admin', 'user', 'guest'], f"Invalid role: {user_dict['role']}"

    # Process based on role...
    if user_dict['role'] == 'admin':
        return "Granting admin privileges"
    elif user_dict['role'] == 'user':
        return "Granting standard user privileges"
    else:  # role is 'guest'
        return "Granting guest privileges"

# This works
print(process_user_data({'name': 'Alice', 'role': 'admin'}))  # Output: Granting admin privileges

# This would fail the assertion
try:
    print(process_user_data({'name': 'Bob', 'role': 'superuser'}))
except AssertionError as e:
    print(e)  # Output: Invalid role: superuser
```

16 Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data in the form of attributes, and code in the form of methods. Below are some key concepts of OOP:

- **Class**: Blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).
- **Object**: An instance of a class.
- **Attribute**: A data element contained within a class or instance.
- **Method**: A function defined within a class.
- **Abstraction**: The concept of hiding the complex implementation details and showing only the necessary features.
- **Encapsulation**: The bundling of data with the methods that operate on that data.
- **Inheritance**: The ability to define a class that inherits all the methods and properties from another class.
- **Polymorphism**: The ability to present the same interface for differing underlying forms.

Before we go inside the technical details, consider a question below:

Describe the action below using Objected Oriented Programming(OOP) and Procedural Oriented Programming (POP) seperately.

`Put an elephant into a refrigerator.`

16.1 Classes and Objects in Python

In Python, a class is defined using the `class` keyword:

```
class Student:
    """A simple class to represent a student."""

    def __init__(self, name, major):
        """Initialize the person with a name and major."""
        self.name = name
        self.major = major

    def greet(self):
        """Print a greeting from the student."""
        print(f"Hello, my name is {self.name} and I majored in {self.major}.")
```

Once a class is defined, we can create objects (instances) of that class:

```
# Create a Student object
Mike = Student("Mike", "CS")

# Access attributes
print(Mike.name)
print(Mike.major)

# Call methods
print(Mike.greet())
```

16.2 The `__init__` Method

The `__init__` method is a special method in Python classes, known as a constructor. It is automatically called when a new object is created:

```
class Student:
    """A simple class to represent a student."""

```

```
    def __init__(self, name, major):
        """Initialize the person with a name and major."""
        self.name = name
        self.major = major

    def greet(self):
        """Print a greeting from the student."""
        print(f"Hello, my name is {self.name} and I majored in {self.major}.")

#This is where the __init__ function is called
Mike = Student("Mike", "CS")
```

The first parameter of a method in a class is conventionally named `self`. **It represents the instance of the class and is used to access variables and methods within the class**:

```
class Student:
    """A simple class to represent a student."""

    def __init__(self, name, major):
        """Initialize the person with a name and major."""
        self.name = name
        self.major = major

    def greet(self):
        """Print a greeting from the student."""
        print(f"Hello, my name is {self.name} and I majored in {self.major}.")

#This is where the __init__ function is called
Mike = Student("Mike", "CS")

Mike.greet() # it is equal to Student.greet(Mike)
Student.greet(Mike)
```

The `self` parameter does not need to be passed by the developer. The Python interpreter will automatically pass the current object reference as `self`.

16.3 Class Variables vs. Instance Variables

Instance variables are variables that are defined inside a method and belong to the instance of a class:

```
class Student:
    """A simple class to represent a student."""

    def __init__(self, name, major):
        """Initialize the person with a name and major."""
        self.name = name      #Instance variable
        self.major = major    #Instance variable

    def greet(self):
        """Print a greeting from the student."""
        print(f"Hello, my name is {self.name} and I majored in {self.major}.")


Mike = Student("Mike", "CS")

# Access instance variables
print(Mike.name)
print(Mike.major)
```

Class variables are variables that are defined within a class but outside any of the class's methods. They are shared among all instances of a class:

```python
class Student:
    """A simple class to represent a student."""
    college = "Wabash College"

    def __init__(self, name, major):
        """Initialize the person with a name and major."""
        self.name = name      #Instance variable
        self.major = major    #Instance variable

    def greet(self):
        """Print a greeting from the student."""
        print(f"Hello, my name is {self.name} and I majored in {self.major}.")

Mike = Student("Mike", "CS")

# Access class variables
print(Student.college)
```

16.4 Elephant Example

Let's write a Python program to illustrate elephant example we discussed before:

```python
import time  # Add this import for time.sleep() function


class Elephant:
    def __init__(self, weight, height, age):  # Should be __init__ with double underscores
        self.weight = weight
        self.height = height
        self.age = age

    def eat(self, amount):
        print("The elephant is eating food...")
        for i in range(amount):
            print("\r{}%".format(100 * (i + 1) / amount), end='')  # Removed extra space after end=
            time.sleep(1)
        print("\nFinished.")  # Added newline for better formatting

    def walk(self, distance, direction):
        for i in range(distance + 1):
            print("\rThe elephant is walking {} : {} feet".format(direction, i),
                  end='')  # Removed extra space before \r
            time.sleep(0.5)
        print("\nFinished.")  # Added newline for better formatting

    def sleep(self, n_hours):
        for i in range(n_hours + 1):
            print("\rThe elephant is sleeping ... {} hour(s)".format(i), end='')
            time.sleep(1)
        print("\nWake up.")  # Added newline and capitalized first letter


class Refrigerator:
    def __init__(self, brand):
        self.status = "close"
        self.food = []
        self.brand = brand

    def open(self):
        self.status = "open"

    def close(self):
        self.status = "close"
```

```
42
43      def put_in(self, food):
44          try:
45              if self.status == "close":
46                  raise ValueError("You must open the door first")
47              #assert self.status == "close", "You must open the door first"
48              elif self.status == "open":
49                  self.foods.append(food)
50              else:
51                  raise ValueError("Unrecognized status!")
52          except Exception as e:
53              print(e)
54          finally:
55              self.close()
56
57      def take_out(self, food):
58          try:
59              assert food in self.food
60          except AssertionError:
61              print("There is no {} in refrigerator!".format(food))
62          else:
63              self.food.remove(food)
64
65  e = Elephant(10,3,5)
66  r = Refrigerator("LG")
67  r.open()
68  r.put_in(e)
69  r.close()
70  print(r.food)
```

16.5 Special Methods (Magic Methods)

Python classes can implement special methods (also called magic methods) that allow custom behavior for built-in operations:

```
1   import copy
2   class Matrix:
3       """A class representing a 2D mathematical matrix."""
4
5       def __init__(self, data):
6           """Initialize a matrix with 2D data.
7
8           Args:
9               data (list): A list of lists representing rows and columns
10          """
11
12          if not data:
13              raise ValueError("Matrix cannot be empty")
14
15          # Validate the input data
16          if not isinstance(data, list) or not all(isinstance(row, list) for row in data):
17              raise TypeError("Matrix data must be a list of lists")
18
19          # Ensure all rows have the same length
20          row_length = len(data[0])
21          if not all(len(row) == row_length for row in data):
22              raise ValueError("All rows must have the same length")
23
24          self.data = copy.deepcopy(data)  # Deep copy of the data
25          self.rows = len(data)
26          self.cols = len(data[0])
27          self.current_row = 0  # For iteration
28
29      def __str__(self):
```

```
        """Return a human-readable string representation."""
        # Format the matrix as a grid
        rows_str = []
        for row in self.data:
            rows_str.append(' '.join(f"{x:4}" for x in row))
        return '\n'.join(rows_str)

    def __len__(self):
        """Return the total number of elements in the matrix."""
        return self.rows * self.cols

    def __add__(self, other):
        """Add two matrices element-wise."""
        if not isinstance(other, Matrix):
            raise TypeError("Can only add with another Matrix")

        if self.rows != other.rows or self.cols != other.cols:
            raise ValueError("Matrices must have the same dimensions for addition")

        result = []
        for i in range(self.rows):
            row = [self.data[i][j] + other.data[i][j] for j in range(self.cols)]
            result.append(row)

        return Matrix(result)

    def __eq__(self, other):
        """Check if two matrices have the same values."""
        if not isinstance(other, Matrix):
            return False

        if self.rows != other.rows or self.cols != other.cols:
            return False

        return all(
            self.data[i][j] == other.data[i][j]
            for i in range(self.rows)
            for j in range(self.cols)
        )

    def __getitem__(self, key):
        """Get element or a single row using indexing."""
        if isinstance(key, tuple) and len(key) == 2:
            row_idx, col_idx = key

            # Single element access
            if isinstance(row_idx, int) and isinstance(col_idx, int):
                if 0 <= row_idx < self.rows and 0 <= col_idx < self.cols:
                    return self.data[row_idx][col_idx]
                raise IndexError("Matrix indices out of range")


        elif isinstance(key, int):
            # Get a single row
            if 0 <= key < self.rows:
                return self.data[key]
            raise IndexError("Row index out of range")

        raise TypeError("Invalid index type for Matrix")

    def __setitem__(self, key, value):
        """Set matrix elements."""
        if isinstance(key, tuple) and len(key) == 2:
            row_idx, col_idx = key

            # Set a single element
```

```
            if isinstance(row_idx, int) and isinstance(col_idx, int):
                if 0 <= row_idx < self.rows and 0 <= col_idx < self.cols:
                    if not isinstance(value, (int, float)):
                        raise TypeError("Matrix elements must be numbers")
                    self.data[row_idx][col_idx] = value
                else:
                    raise IndexError("Matrix indices out of range")
            else:
                raise TypeError("Setting slices not supported")

        elif isinstance(key, int):
            # Set a whole row
            if 0 <= key < self.rows:
                if not isinstance(value, list) or len(value) != self.cols:
                    raise ValueError(f"Expected a list of length {self.cols}")
                if not all(isinstance(x, (int, float)) for x in value):
                    raise TypeError("Matrix elements must be numbers")
                self.data[key] = copy.deepcopy(value)
            else:
                raise IndexError("Row index out of range")

        else:
            raise TypeError("Invalid index type for Matrix")

    def __iter__(self):
        """Return an iterator for the matrix rows."""
        self.current_row = 0
        return self

    def __next__(self):
        """Return the next row when iterating."""
        if self.current_row >= self.rows:
            raise StopIteration

        row = self.data[self.current_row]
        self.current_row += 1
        return row
```

- `__init__`: Constructor
- `__str__`: String representation (for humans)
- `__len__`: Length
- `__add__`, `__sub__`, `__mul__`, etc.: Arithmetic operations
- `__eq__`, `__lt__`, `__gt__`, etc.: Comparison operations
- `__getitem__`, `__setitem__`: Indexing operations
- `__iter__`, `__next__`: Iteration

16.6 Privacy in Python Class

In Python, by convention, a name prefixed with an underscore (e.g., `_name`) should be treated as a non-public part of the API.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance  # "private" attribute

    def deposit(self, amount):
        self._balance += amount
        return self._balance

    def withdraw(self, amount):
```

```
        if amount <= self._balance:
            self._balance -= amount
            return True
        return False

    def get_balance(self):
        return self._balance

account = BankAccount("Alice", 1000)

# Accessing the "private" attribute directly (not recommended)
print(account._balance)  # Output: 1000

# Using the public method to access the "private" attribute
print(account.get_balance())  # Output: 1000
```

For a stronger form of privacy, Python supports name mangling. Names with double leading underscores (e.g., `__name`) are mangled with the class name:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # Private attribute with name mangling

    def deposit(self, amount):
        self.__balance += amount
        return self.__balance

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
            return True
        return False

    def get_balance(self):
        return self.__balance

account = BankAccount("Alice", 1000)
# This will raise an AttributeError
# print(account.__balance)
# The attribute is actually stored as _BankAccount__balance
print(account._BankAccount__balance)  # Output: 1000
# Using the public method to access the private attribute
print(account.get_balance())  # Output: 1000
```

16.7 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. A class inherits from a single parent class, as shown below:

```
# Parent class
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        return "Some generic animal sound"

# Child class
class Dog(Animal):
    def __init__(self, name, breed):
```

```
        # Call the parent class's __init__ method
        super().__init__(name, species="Canis familiaris")
        self.breed = breed

    def make_sound(self):
        return "Woof!"

# Child class
class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Felis catus")
        self.breed = breed

    def make_sound(self):
        return "Meow!"

dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Siamese")

print(dog.name)         # Output: Buddy
print(dog.species)      # Output: Canis familiaris
print(dog.make_sound())  # Output: Woof!

print(cat.name)         # Output: Whiskers
print(cat.species)      # Output: Felis catus
print(cat.make_sound())  # Output: Meow!
```

A class inherits from multiple parent classes:

```
class Swimmer:
    def __init__(self):
        super().__init__()
        print("Swimmer initialized")

    def swim(self):
        print(f"Swimming")

    def dive(self):
        print("Diving")


class Flyer:
    def __init__(self):
        super().__init__()
        print("Flyer initialized")

    def fly(self):
        print("Flying")

    def land(self):
        print("Landing")


class Duck(Swimmer, Flyer):
    def __init__(self):
        # Call the parent class initializers
        super().__init__()
        print(f"A duck is initialized")

    def quack(self):
        print("Quack quack!")

    # Override the swim method from Swimmer
    def swim(self):
```

```
        print(f"Paddling efficiently")


# Create a duck instance
d1 = Duck()
d1.swim()
d1.fly()
```

When a class inherits from multiple parent classes, Python uses the Method Resolution Order (MRO) to determine which method to call. The MRO is a built-in mechanism that determines the order in which base classes are searched when looking for a method:

```
class A:
    def method(self):
        return "Method from A"

class B(A):
    def method(self):
        return "Method from B"

class C(A):
    def method(self):
        return "Method from C"

class D(B, C):
    pass

d = D()
print(d.method())  # Output: Method from B

# View the MRO
print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

In Python, `<class 'object'>` refers to the base class for all Python classes. **It's the ultimate parent class in Python's inheritance hierarchy**, meaning every class implicitly inherits from object if no explicit parent class is specified. When you examine a class's Method Resolution Order (MRO) using Class.__mro__, `object` **always appears as the last class in the sequence**.

It contains several default methods with default implementations like:

__new__: For creating instances __init__: For initializing instances __str__ and __repr__: For string representations __eq__, __hash__: For comparison and hashing

Specifically, you can understand __new__ and __init__ as below(in fact, the object class is written in C as the fundamental of Python, check object.c to see more details if you are interested):

```
class object:
    def __new__(cls, *args, **kwargs):
        """Create and return a new instance of the class."""
        # The actual implementation allocates memory for a new instance
        # and returns the newly created object
        instance = allocate_memory_for_instance(cls)
        return instance

    def __init__(self, *args, **kwargs):
        """Initialize the instance."""
        # The base object.__init__ does nothing
        pass
```

16.7.1 The `super()` Function

The `super()` function provides a way to access methods and attributes from parent classes. It returns a proxy object that delegates method calls to the next class in the Method Resolution Order (MRO).

In Python 3, within an instance method, you can simply use `super()` without arguments, which is equivalent to `super(CurrentClass, self)`.

```
class Parent:
    def __init__(self, value):
        self.value = value

    def greet(self):
        return f"Hello from Parent with value {self.value}"

class Child(Parent):
    def __init__(self, value, extra):
        super().__init__(value)  # Call Parent's __init__
        self.extra = extra

    def greet(self):
        return f"{super().greet()} and extra {self.extra}"

# Create a Child instance
child = Child(42, "extra data")
print(child.greet())
# Output: Hello from Parent with value 42 and extra extra data
```

The behavior of `super()` becomes more complex with multiple inheritance. MRO uses the order in which parent classes are searched for attributes.

```
class A:
    def method(self):
        return "A.method"

class B(A):
    def method(self):
        return f"B.method -> {super().method()}"

class C(A):
    def method(self):
        return f"C.method -> {super().method()}"

class D(B, C):
    def method(self):
        return f"D.method -> {super().method()}"

# Create a D instance
d = D()
print(d.method())

# Print the MRO
print(D.__mro__)
```

Output:

```
D.method -> B.method -> C.method -> A.method
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
    <class '__main__.A'>, <class 'object'>)
```

In this example, `super()` in `D.method` calls `B.method`, `super()` in `B.method` calls `C.method`, and `super()` in `C.method` calls `A.method`. This is determined by the MRO.

When to Use `super()`?

1. **Calling the Parent Constructor**:The most common use of `super()` is to call the parent class's `__init__` method:

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call parent's __init__
        self.breed = breed
```

2. **Method Extension**:Use `super()` to extend a parent class's method rather than completely overriding it. The above `__init__` is an example.

3. **Cooperative Multiple Inheritance**: `super()` is essential for cooperative multiple inheritance, where each class in the inheritance chain needs to be properly initialized:

```
class Base:
    def __init__(self):
        print("Base.__init__")

class A(Base):
    def __init__(self):
        print("A.__init__")
        super().__init__()

class B(Base):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A, B):
    def __init__(self):
        print("C.__init__")
        super().__init__()

# Create a C instance
c = C()
print(C.__mro__)
```

Output:

```
C.__init__
A.__init__
B.__init__
Base.__init__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
    <class '__main__.Base'>, <class 'object'>)
```

This example demonstrates the "super() call chain" where each class calls `super().__init__()`, ensuring that **all classes in the inheritance hierarchy are properly initialized exactly once**.

If you manually initialize the parent classes:

```
class Base:
    def __init__(self):
        print("Base.__init__")


class A(Base):
    def __init__(self):
        print("A.__init__")
```

```
        Base.__init__(self)
        # super().__init__()


class B(Base):
    def __init__(self):
        print("B.__init__")
        Base.__init__(self)
        # super().__init__()


class C(A, B):
    def __init__(self):
        print("C.__init__")
        A.__init__(self)
        B.__init__(self)


# Create a C instance
c = C()
print(C.__mro__)
```

Output:

```
C.__init__
A.__init__
Base.__init__
B.__init__
Base.__init__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
    <class '__main__.Base'>, <class 'object'>)
```

It could lead to a more fatal error if you do not use `super()` properly:

```
class Swimmer:
    def __init__(self):
        print("Swimmer initialized")
        self.swimming_speed = 10
        self.can_swim = True

    def swim(self):
        print(f"Swimming at speed {self.swimming_speed}")

class Flyer:
    def __init__(self):
        print("Flyer initialized")
        self.flying_speed = 20
        self.can_fly = True

    def fly(self):
        print(f"Flying at speed {self.flying_speed}")

# Problematic initialization
class Duck(Swimmer, Flyer):
    def __init__(self):
        # Only calls Swimmer.__init__, not Flyer.__init__
        super().__init__()
        print("Duck initialized")

    def travel(self):
        print(f"Duck can swim: {self.can_swim}")
        print(f"Duck can fly: {self.can_fly}")  # AttributeError will occur here
        print(f"Total speed: {self.swimming_speed + self.flying_speed}")

# Try to use the Duck
```

```python
try:
    duck = Duck()
    duck.swim()  # Works fine
    duck.fly()   # Will try to use flying_speed attribute that was never set
    duck.travel() # Will definitely fail
except AttributeError as e:
    print(f"Error occurred: {e}")
```

Output:

```
Swimmer initialized
Duck initialized
Swimming at speed 10
Error occurred: 'Duck' object has no attribute 'flying_speed'
```

The Duck object was only partially initialized. It received attributes from the Swimmer class but none from the Flyer class, making any method that relies on Flyer's attributes fail.

16.7.2 `__new__` method

Before diving into `__new__`, it's important to understand the object creation process in Python:

1. When you call a class (e.g., `x = MyClass()`), Python first calls `__new__` to create an instance.
2. After `__new__` creates the instance, `__init__` is called to initialize it.

This **two-step process** separates object creation from object initialization, which enables some advanced programming patterns.

`__new__` is a **static method** that is responsible for creating and returning a new instance of a class. It is called before `__init__` and is the true constructor in Python's object model.

The signature of `__new__` is:

```python
def __new__(cls, *args, **kwargs):
    # Create and return an instance
```

Key points about `__new__`:

- It receives the class (`cls`) as its first argument, not the instance.
- It must return an instance, typically of the class `cls`.
- It's a static method, but doesn't require the `@staticmethod` decorator because Python handles it specially.
- If `__new__` returns an instance of `cls`, then `__init__` is automatically called on that instance.
- If `__new__` returns something that is not an instance of `cls`, then `__init__` is not called.

Here's a simple example of a class with a custom `__new__` method:

```python
class MyClass:
    def __new__(cls, *args, **kwargs):
        print(f"__new__ called with class {cls.__name__}")
        instance = super().__new__(cls)
        return instance

    def __init__(self, value):
        print(f"__init__ called with value {value}")
        self.value = value

# Create an instance
obj = MyClass(42)
print(f"obj.value = {obj.value}")
```

You can use `__new__` to control how instances are created, including implementing object pools or preventing instantiation under certain conditions:

```python
class RestrictedClass:
    _allowed_values = [1, 2, 3, 4, 5]

    def __new__(cls, value):
        if value not in cls._allowed_values:
            raise ValueError(f"Value must be one of {cls._allowed_values}")
        return super().__new__(cls)

    def __init__(self, value):
        self.value = value

# This works
obj = RestrictedClass(3)
print(obj.value)  # 3

# This raises a ValueError
# obj = RestrictedClass(10)
```

__new__ can also be used to create immutable objects by returning existing instances for equivalent values:

```python
class Immutable:
    _instances = {}

    def __new__(cls, value):
        if value in cls._instances:
            return cls._instances[value]

        instance = super().__new__(cls)
        cls._instances[value] = instance
        return instance

    def __init__(self, value):
        # This check is needed since __init__ is called each time
        if not hasattr(self, 'value'):
            self.value = value

# Create instances
a = Immutable(5)
b = Immutable(5)
c = Immutable(10)

print(a is b)  # True - same instance
print(a is c)  # False - different instance
```

16.8 Polymorphism

Polymorphism allows methods to be used in different ways for different data inputs, we've already seen this before:

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

```

```
class Duck(Animal):
    def speak(self):
        return "Quack!"

# Function that demonstrates polymorphism
def animal_sound(animal):
    return animal.speak()

# Create instances of different animals
dog = Dog()
cat = Cat()
duck = Duck()

# Call the same method on different objects
print(animal_sound(dog))   # Output: Woof!
print(animal_sound(cat))   # Output: Meow!
print(animal_sound(duck))  # Output: Quack!
```

16.9 Let's Do An Exercise

Design an object-oriented program to simulate the process of making a pizza and cooking it in an oven.

- **Pizza Class:**
  - Requires ingredients to initialize a pizza object.
  - Must include `flour` and `cheese` as mandatory ingredients.
  - Methods:
    - `__new__()`
    - `__init__()`
    - `__str__()`
    - `__del__()`
    - `make()`
  - Attribute:
    - `name` (stores the name of the pizza)
- **Oven Class:**
  - Attributes:
    - Requires `brand` and `dimensions` for initialization.
    - `dimensions` must be a list of length 3.
    - `door` (records whether the oven door is open or closed)
    - `occupied` (records what is inside the oven)
  - Methods:
    - `__new__()`
    - `__init__()`
    - `__str__()`
    - `__del__()`
    - `open()`
    - `close()`
    - `put_in()`
    - `take_out()`
    - `cook()`

– The `cook()` method should check whether there is anything in the oven and only proceed if a pizza is inside and the oven door is closed.

17 Decorators

Python decorators allow you to modify or extend the behavior of functions and methods without changing their actual code. When you use a Python decorator, you wrap a function with another function, which takes the original function as an argument and returns its modified version. This technique provides a simple way to implement higher-order functions in Python, enhancing code reusability and readability.

17.1 Review Functions in Python

While functions have various attributes, in the context of decorators, it is essential to recognize that **a function transforms given arguments into a return value**. Consider this elementary example:

```
>>> def add_one(number):
...     return number + 1
...

>>> add_one(2)
3
```

Generally, Python functions may produce side effects beyond simply converting inputs to outputs. The `print()` function exemplifies this behavior by returning `None` while simultaneously generating console output. However, for understanding decorators, it suffices to conceptualize functions as mechanisms that convert arguments into values.

Python treats functions as **first-class objects**. This designation means that **functions can be manipulated and passed as arguments**, similar to other objects like `str`, `int`, `float`, `list`, etc. Consider these three function definitions:

```
def say_hello(name):
    return f"Hello {name}"

def be_awesome(name):
    return f"Yo {name}, together we're the awesomest!"

def greet_bob(greeter_func):
    return greeter_func("Bob")

print(greet_bob(say_hello)) # Hello Bob
```

In this example, `say_hello()` and `be_awesome()` are conventional functions expecting a string name parameter. The `greet_bob()` function, however, expects a function as its parameter. You can supply either `say_hello()` or `be_awesome()` as an argument.

Note that `greet_bob(say_hello)` references two distinct functions—`greet_bob()` and `say_hello`—in different contexts. The `say_hello` function is referenced without parentheses, indicating that only a function reference is transmitted without execution. Conversely, `greet_bob()` includes parentheses, signaling normal function invocation.

This distinction is crucial for understanding functions as first-class objects. A function name without parentheses constitutes a function reference, while a function name with trailing parentheses invokes the function and references its return value.

17.1.1 Inner Functions

Functions can be defined within other functions; these are termed inner functions. Below is an example featuring a function with two inner functions:

```
def parent():
    print("Printing from parent()")

    def first_child():
        print("Printing from first_child()")

    def second_child():
        print("Printing from second_child()")
```

```
    second_child()
    first_child()
```

What occurs when the `parent()` function is called? The output appears as:

```
Printing from parent()
Printing from second_child()
Printing from first_child()
```

Inner functions aren't defined until the parent function is invoked. They exist within the local scope of `parent()`, functioning as local variables. Attempting to call `first_child()` directly produces an error:

```
>>> first_child()
Traceback (most recent call last):
  ...
NameError: name 'first_child' is not defined
```

When `parent()` is called, the inner functions `first_child()` and `second_child()` are also invoked. However, due to their local scope, they remain inaccessible outside the `parent()` function.

17.1.2 Functions as Return Values

Python allows functions to return other functions. In the following example, `parent()` is modified to return one of its inner functions:

```
def parent(num):
    print("Printing from parent()")

    def first_child(c1):
        print(f"Printing from first_child {c1}")

    def second_child(c2):
        print(f"Printing from second_child {c2}")

    if num == 1:
        return first_child
    else:
        return second_child
```

Note that `first_child` is returned without parentheses, indicating that a reference to the `first_child` function is being returned. In contrast, `first_child()` with parentheses would represent the result of evaluating the function. This distinction is illustrated below:

```
>>> first = parent(1)
>>> second = parent(2)

>>> first
<function parent.<locals>.first_child at 0x7f599f1e2e18>

>>> second
<function parent.<locals>.second_child at 0x7f599dad5268>
```

The cryptic output indicates that `first` references the local `first_child()` function within `parent()`, while `second` references `second_child()`.

These variables can now be utilized as standard functions, despite the inability to directly access the functions they reference:

```
>>> first("name1")
'Printing from first_child name1'
```

The output confirms these are indeed the return values from the inner functions defined within `parent()`.

17.2 Simple Decorators in Python

Let's begin with an example:

```
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = decorator(say_whee)
```

Here, we've defined two regular functions, `decorator()` and `say_whee()`, and one inner `wrapper()` function. Then we redefined `say_whee()` to apply `decorator()` to the original `say_whee()`.

When you call `say_whee()`, this is what happens:

```
say_whee()

Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

To understand what's happening, let's examine the decoration line:

```
say_whee = decorator(say_whee)
```

Effectively, the name `say_whee` now points to the `wrapper()` inner function. Remember that we return `wrapper` as a function when we call `decorator(say_whee)`:

```
print(say_whee)

<function decorator.<locals>.wrapper at 0x7f3c5dfd42f0>
```

However, `wrapper()` has a reference to the original `say_whee()` as `func`, and it calls that function between the two calls to `print()`.

In simple terms, a decorator wraps a function, modifying its behavior.

Because `wrapper()` is a regular Python function, the way a decorator modifies a function can change dynamically. The following example will only run the decorated code during the day:

```
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
```

```
        else:
            pass  # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
```

If you try to call `say_whee()` after bedtime, nothing will happen.

The way we decorated `say_whee()` in our first example is somewhat clunky. We have to type the name `say_whee` three times, and the decoration gets hidden away below the function definition.

Python provides a more elegant way to use decorators with the `@` symbol, sometimes called the "pie syntax":

```
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@decorator
def say_whee():
    print("Whee!")
```

So, `@decorator` is just a shorter way of saying `say_whee = decorator(say_whee)`. It's how you apply a decorator to a function.

Since a decorator is just a regular Python function, all the usual tools for reusability are available. Let's create a module to store our decorators:

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

The `do_twice()` decorator calls the decorated function twice:

```
@do_twice
def say_whee():
    print("Whee!")

say_whee()

>>>Whee!
>>>Whee!
```

17.2.1 Decorating Functions With Arguments

What if you have a function that accepts arguments? Can you still decorate it? Let's try:

```
@do_twice
def greet(name):
    print(f"Hello {name}")
```

Unfortunately, calling this function raises an error:

```
greet("World")


Traceback (most recent call last):
  ...
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

The problem is that the inner function `wrapper_do_twice()` doesn't accept any arguments, but we passed `name="World"` to it. We could fix this by letting `wrapper_do_twice()` accept one argument, but then it wouldn't work for the `say_whee()` function created earlier.

The solution is to use `*args` and `**kwargs` in the inner wrapper function to accept an arbitrary number of positional and keyword arguments:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

The `wrapper_do_twice()` inner function now accepts any arguments and passes them to the decorated function. Now both our `say_whee()` and `greet()` examples work:

```
@do_twice
def greet(name):
    print(f"Hello {name}")

@do_twice
def say_whee():
    print("Whee!")

say_whee()
greet("World")

>>>Whee!
>>>Whee!
>>>Hello World
>>>Hello World
```

We use the same decorator, `@do_twice`, to decorate two different functions. This demonstrates one of the powers of decorators: **they add behavior that can apply to many different functions.**

17.2.2 Returning Values From Decorated Functions

What happens to the return value of decorated functions? That's up to the decorator to decide. Consider this example:

```
@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

When we try to use it:

```
hi_adam = return_greeting("Adam")
print(hi_adam)

```

```
>>> Creating greeting
>>> Creating greeting
>>> None
```

Because `wrapper_do_twice()` doesn't explicitly return a value, the call `return_greeting("Adam")` returns `None`.

To fix this, we need to ensure the wrapper function returns the return value of the decorated function:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs) # first evaluate func() then return its return value
    return wrapper_do_twice
```

Now we return the value from the last call of the decorated function:

```
hi_adam = return_greeting("Adam")
print(hi_adam)

Creating greeting
Creating greeting
'Hi Adam'
```

17.2.3 Preserving Function Identity

Python's introspection allows objects to know about their own attributes at runtime. For instance, a function knows its own name and documentation:

```
>>> print.__name__
'print'

>>> help(print)
Help on built-in function print in module builtins:

print(...)
    <full help message>
```

However, after decoration, our function has an identity crisis:

```
>>> say_whee
<function do_twice.<locals>.wrapper_do_twice at 0x7f43700e52f0>

>>> say_whee.__name__
'wrapper_do_twice'

>>> help(say_whee)
Help on function wrapper_do_twice in module decorators:

wrapper_do_twice()
```

To fix this, decorators should use the `@functools.wraps` decorator, which preserves information about the original function:

```
import functools

def do_twice(func):
    @functools.wraps(func)
```

```
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

Much better! Now `say_whee()` maintains its original identity after decoration.

17.3 A Few Real World Examples

We'll now examine several practical examples of decorators. You'll notice they generally follow the pattern you've already learned:

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

This structure serves as an excellent template for developing more sophisticated decorators.

17.3.1 Timing Functions

Let's begin by implementing a `@timer` decorator that measures a function's execution time and outputs the duration to the console:

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__}() in {run_time:.4f} secs")
        return value
    return wrapper_timer
```

This decorator operates by recording the time immediately before function execution (line 10) and just after completion (line 12). The function's runtime is calculated as the difference between these measurements (line 13). We utilize `time.perf_counter()`, which effectively measures time intervals.

To test our `@timer`, we'll create a `waste_some_time()` function that performs time-consuming operations:

```
@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([number**2 for number in range(10_000)])

waste_some_time(1)
waste_some_time(999)

>>>Finished waste_some_time() in 0.0016 secs
>>>Finished waste_some_time() in 1.5771 secs
```

17.3.2 Debugging Code

The following `@debug` decorator displays a function's arguments and return value each time it's invoked:

```
1  import functools
2
3  def debug(func):
4      """Print the function signature and return value"""
5      @functools.wraps(func)
6      def wrapper_debug(*args, **kwargs):
7          args_repr = [repr(a) for a in args]
8          kwargs_repr = [f"{k}={repr(v)}" for k, v in kwargs.items()]
9          signature = ", ".join(args_repr + kwargs_repr)
10         print(f"Calling {func.__name__}({signature})")
11         value = func(*args, **kwargs)
12         print(f"{func.__name__}() returned {repr(value)}")
13         return value
14     return wrapper_debug
```

The function signature is constructed by concatenating string representations of all arguments:

- **Line 7:** Creates a list of positional arguments, using `repr()` to generate a string representation for each.
- **Line 8:** Creates a list of keyword arguments, formatting each as `key=value` with `repr()` for value representation.
- **Line 9:** Joins the positional and keyword argument lists into a single signature string, with arguments separated by commas.
- **Line 12:** Prints the return value after function execution.

Let's see the decorator in action by applying it to a simple function with one positional and one keyword argument:

```
1  @debug
2  def make_greeting(name):
3          return f"Hi {name}!"
```

Observe how the `@debug` decorator outputs the signature and return value of `make_greeting()`:

```
1  make_greeting("Dr. Deng")
2
3  >>>Calling make_greeting('Dr. Deng')
4  >>>make_greeting() returned 'Hi Dr. Deng!'
```

While this example might not appear immediately useful since `@debug` merely echoes your input, it becomes powerful when applied to utility functions that you don't directly call.

The following example calculates an approximation of the mathematical constant e:

```
1  import math
2
3  math.factorial = debug(math.factorial)
4
5  def approximate_e(terms=18):
6      return sum(1 / math.factorial(n) for n in range(terms))
```

Here, we apply a decorator to a previously defined function. In line 3, we decorate `factorial()` from the `math` standard library. While we can't use the pie syntax, we can manually apply the decorator. The approximation of e is based on the series expansion:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

When invoking the `approximate_e()` function, we can observe the `@debug` decorator in action:

```
print(approximate_e(terms=5))

>>>Calling factorial(0)
>>>factorial() returned 1
>>>Calling factorial(1)
>>>factorial() returned 1
>>>Calling factorial(2)
>>>factorial() returned 2
>>>Calling factorial(3)
>>>factorial() returned 6
>>>Calling factorial(4)
>>>factorial() returned 24
>>>2.708333333333333
```

In this example, we obtain a reasonable approximation of $e \approx 2.718281828$ by summing just five terms.

17.3.3 Slowing Down Code

While intentionally slowing down code might seem counterintuitive, it serves several practical purposes. The most prevalent application involves rate-limiting functions that repeatedly check for changes in resources such as web pages. Our `@slow_down` decorator will introduce a one-second pause before executing the decorated function:

```
import functools
import time

def slow_down(func):
    """Sleep 1 second before calling the function"""
    @functools.wraps(func)
    def wrapper_slow_down(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper_slow_down
```

In the `@slow_down` implementation, we invoke `time.sleep()` to create a delay before executing the decorated function. To demonstrate this decorator's effect, let's create a `countdown()` function and observe the result:

```
@slow_down
def countdown(from_number):
    if from_number < 1:
        print("Liftoff!")
    else:
        print(from_number)
        countdown(from_number - 1)

>>> countdown(3)
3
2
1
Liftoff!
```

The `countdown()` function checks if `from_number` is less than one. If true, it prints "Liftoff!"; otherwise, it displays the current number and continues counting down. **Note:** `countdown()` is a recursive function—it calls itself.

17.4 Fancy Decorators

Decorators can interact with classes in two distinct ways. The first approach parallels function decoration: applying decorators to methods within a class. This capability was a primary motivation for the introduction of decorators in Python.

Several decorators are integrated into Python's core functionality, including `@classmethod`, `@staticmethod`, and `@property`. The `@classmethod` and `@staticmethod` decorators facilitate the definition of methods within a class namespace that operate independently of specific class instances. The `@property` decorator enables customized access and modification of class attributes.

17.4.1 Decorating Classes

```
class Circle:
    def __init__(self, radius):
        self._radius = radius
        self._area = None

    @property #Instead of always recalculating the area, it now only calculates it if self._area is None.
    def area(self):
        # Only calculate if _area is None
        if self._area is None:
            self._area = self.pi() * self._radius ** 2
        return self._area

    @area.setter
    def area(self, value):
        # Just set the value directly
        self._area = value

    def cylinder_volume(self, height):
        return self.area * height

    @classmethod
    def unit_cir(cls):
        return Circle(1)

    @staticmethod
    def pi():
        return 3.14


c1 = Circle(2)
print(c1.pi())
print(Circle.pi())
print(c1.area)  # This will calculate the area
c1.area = 10  # This will manually set the area
print(c1.area)  # This will return the manually set value
```

The `Circle` class demonstrates various method types distinguished by decorators:

- `cylinder_volume()` represents a conventional instance method.
- `area` function should be immutable; here, for illustration purposes, if you want to make it mutable, you have to implement a setter function.
- `unit_circle()` operates as a class method unbound to specific instances, commonly employed as a factory method. (the first parameter is cls not self)
- `pi()` serves as a static method, independent of the class except for namespace inclusion, callable from either instances or the class.(The first parameter is neither cls nor self)

When applying function decorators to classes, the effects may diverge from expectations:

```
@timer
class TimeWaster:
    def __init__(self, max_num):
        self.max_num = max_num

    def waste_time(self, num_times):
```

```
        for _ in range(num_times):
            sum([i**2 for i in range(self.max_num)])
```

Important distinction: decorating a class doesn't decorate its constituent methods. The `@timer` decorator only measures instantiation time:

```
tw = TimeWaster(1000)
>>>Finished TimeWaster() in 0.0000 secs

tw.waste_time(999)
```

The output confirms that only class instantiation is timed, while the `waste_time()` method execution remains unmonitored.

17.4.2 Nesting Decorators

Multiple decorators can be applied to a single function by stacking them:

```
@debug
@do_twice
def greet(name):
    print(f"Hello {name}")
```

Decorators are executed in the order they appear. In this example, `@debug` calls `@do_twice`, which then calls `greet()`, equivalent to `debug(do_twice(greet()))`:

```
greet("Dr. Deng")

>>>Calling greet('Dr. Deng')
>>>Hello Dr. Deng
>>>Hello Dr. Deng
>>>greet() returned None
```

The `@do_twice` decorator causes the greeting to be printed twice, while `@debug` output appears just once since it's applied before `@do_twice`. Reversing the decorator order produces different results:

```
@do_twice
@debug
def greet(name):
    print(f"Hello {name}")

>>>Calling greet('Dr. Deng')
>>>Hello Dr. Deng
>>>greet() returned None
>>>Calling greet('Dr. Deng')
>>>Hello Dr. Deng
>>>greet() returned None
```

In this case, `@do_twice` is applied to both `@debug` and the function, resulting in debug information for each call.

17.4.3 Defining Decorators With Arguments

Decorators can accept arguments to customize their behavior. For example, `@do_twice` could be extended to a `@repeat(num_times)` decorator that specifies execution frequency:

```
@repeat(num_times=4)
def greet(name):
    print(f"Hello {name}")

greet("World")

>>>Hello World
>>>Hello World
>>>Hello World
>>>Hello World
```

To implement a parameterized decorator like `@repeat`, we need `repeat(num_times=4)` to return a function object that can act as a decorator. The implementation requires three nested functions:

```
import functools

def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat
    return decorator_repeat
```

The innermost function `wrapper_repeat()` accepts arbitrary arguments and returns the decorated function's value, executing it `num_times` times in a loop.

The middle function `decorator_repeat()` serves as the actual decorator, preserving the original function's metadata with `@functools.wraps`.

The outermost function `repeat()` accepts the decorator's parameter and returns the decorator function. By defining `decorator_repeat()` inside `repeat()`, a closure is created that preserves the `num_times` value until it's needed by `wrapper_repeat()`.

This implementation allows for flexible decorator configuration while maintaining the standard decorator pattern.

17.4.4 Tracking State in Decorators

Decorators can maintain state between function calls, which provides additional functionality beyond simple function transformation. We'll explore how to create a decorator that counts function invocations.

For simpler cases, **function attributes** offer an elegant solution for state tracking. In Python, the term "attribute" is more general than you might be thinking. Attributes can refer to:

1. Class attributes (defined in a class)
2. Instance attributes (defined for objects)
3. Attributes on any Python object, including functions

Python functions are objects themselves, and like any Python object, they can have attributes attached to them. This isn't related to class-based attributes but is rather a feature of Python's object system.

When you write:

```
def my_function():
    pass

my_function.count = 0
```

You're not modifying an instance of a class in the traditional sense. Instead, you're adding an attribute directly to the function object itself.

This works because Python functions are first-class objects. They can be:

- Assigned to variables
- Passed as arguments
- Returned from other functions
- Stored in data structures
- Have attributes assigned to them

We're not modifying a class instance in the OOP sense - we're attaching a new attribute to the function object named `my_function`.

This behavior is different from many other programming languages where functions aren't treated as first-class objects with their own namespaces. It's part of what makes Python flexible and enables patterns like the function attribute tracking we saw in the decorator examples.

```
import functools

def count_calls(func):
    @functools.wraps(func)
    def wrapper_count_calls(*args, **kwargs):
        wrapper_count_calls.num_calls += 1
        print(f"Call {wrapper_count_calls.num_calls} of {func.__name__}()")
        return func(*args, **kwargs)
    wrapper_count_calls.num_calls = 0 # initializing a counter attribute on the wrapper function before returning it
    return wrapper_count_calls
```

The wrapper function stores call count in its `num_calls` attribute. When applied:

```
@count_calls
def say_whee():
    print("Whee!")


say_whee()
say_whee()
print(say_whee.num_calls)

>>>Call 1 of say_whee()
>>>Whee!
>>>Call 2 of say_whee()
>>>Whee!
>>>2
```

Each invocation increments the counter, and we can directly access the `.num_calls` attribute to retrieve the current count.

17.4.5 Using Classes as Decorators

Classes provide a more structured approach to maintaining state in Python. We can reimplement our counting decorator using class-based syntax. For a class instance to function as a callable, we implement the special `__call__` method:

```
class Counter:
    def __init__(self, start=0):
        self.count = start
    def __call__(self):
        self.count += 1
        print(f"Current count is {self.count}")
```

The `__call__` method executes whenever the instance is called:

```
counter = Counter()
counter()
counter()

>>>Current count is 1
>>>Current count is 2
```

A decorator class typically implements both `__init__` and `__call__`:

```
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__}()")
        return self.func(*args, **kwargs)
```

The `__init__` method stores a reference to the decorated function and performs initialization. The `__call__` method replaces the decorated function, similar to wrapper functions in previous examples. Note the use of `functools.update_wrapper()` instead of the `@functools.wraps` decorator.

This class-based decorator functions identically to our previous implementation:

```
@CountCalls
def say_whee():
    print("Whee!")


say_whee()
say_whee()
print(say_whee.num_calls)

>>>Call 1 of say_whee()
>>>Whee!
>>>Call 2 of say_whee()
>>>Whee!
>>>2
```

Each function call is counted and reported, demonstrating how class-based decorators can effectively maintain state.

17.5 More Real-World Examples

17.5.1 Parameterizing the slow_down Decorator

Previously, the `@slow_down` decorator introduced a fixed one-second delay. Now, we can enhance it with an optional parameter to control the delay duration:

```
import functools
import time

def slow_down(_func=None, *, rate=1):
    """Sleep given amount of seconds before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
```

```
            return func(*args, **kwargs)
        return wrapper_slow_down

    if _func is None:
        return decorator_slow_down
    else:
        return decorator_slow_down(_func)
```

The if else block is necessary because:

- When you use @slow_down (no parentheses), Python passes the function being decorated directly as the _func parameter.
- When you use @slow_down(rate=2) (with parentheses), Python first executes slow_down(rate=2), which means _func is None and only the rate parameter is set.

This implementation employs the boilerplate pattern described in the section on creating decorators with optional arguments, allowing `@slow_down` to be used both with and without parameters. The example below demonstrates a recursive countdown function with a two-second pause between iterations:

```
@slow_down(rate=2)
def countdown(from_number):
    if from_number < 1:
        print("Liftoff!")
    else:
        print(from_number)
        countdown(from_number - 1)

countdown(3)
```

When executed, the output appears with a two-second delay between each number.

17.5.2 Creating Singletons

A singleton is a class that permits only one instance. Python includes several built-in singletons such as `None`, `True`, and `False`. The following `@singleton` decorator transforms a class into a singleton by storing the first instance and returning it for all subsequent instantiation attempts:

```
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if wrapper_singleton.instance is None:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton
```

This class decorator follows the same pattern as function decorators, with `cls` replacing `func` as the parameter name to indicate its intended use with classes.

In practice:

```
@singleton
class TheOne:
    pass

first_one = TheOne()
another_one = TheOne()
print(id(first_one))
```

```
print(id(another_one))
print(first_one is another_one)

>>>4342416240
>>>4342416240
>>>True
```

Class decorators are used less frequently than function decorators and should be thoroughly documented to guide proper usage.

17.5.3 Caching Return Values

Decorators offer an elegant approach for implementing caching and memoization. Consider this recursive implementation of the Fibonacci sequence:

```
@count_calls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

While this implementation is conceptually clear, its performance characteristics are problematic:

```
fibonacci(10)
>>>55

print(fibonacci.num_calls)
>>>177
```

Computing the tenth Fibonacci number should theoretically require calculating only the preceding Fibonacci numbers, yet this implementation requires 177 calculations. The situation deteriorates rapidly—calculating `fibonacci(20)` necessitates 21,891 calculations, while `fibonacci(30)` requires nearly 2.7 million calculations. This inefficiency stems from repeatedly recalculating previously determined Fibonacci numbers.

A conventional solution employs a loop with a lookup table. Alternatively, caching provides an effective approach.

The Python standard library provides a Least Recently Used (LRU) cache via `@functools.lru_cache` and a regular cache with `@functools.cache`.

```
@functools.lru_cache(maxsize=4)
def fibonacci(num):
    if num < 2:
        value = num
    else:
        value = fibonacci(num - 1) + fibonacci(num - 2)
    print(f"Calculated fibonacci({num}) = {value}")
    return value
```

The `maxsize` parameter defines the number of recent calls to cache. The default is 128, but specifying `maxsize=None` enables caching of all function calls—equivalent to using `@functools.cache`. However, be cautious as this can lead to memory issues when caching numerous large objects.

The `.cache_info()` method provides insights into cache performance for potential tuning. In this example, we deliberately use a small `maxsize` to demonstrate cache eviction behavior:

```
print(fibonacci(10), fibonacci.cache_info())

>>>Calculated fibonacci(1) = 1
>>>Calculated fibonacci(0) = 0
>>>Calculated fibonacci(2) = 1
```

```
>>>Calculated fibonacci(3) = 2
>>>Calculated fibonacci(4) = 3
>>>Calculated fibonacci(5) = 5
>>>Calculated fibonacci(6) = 8
>>>Calculated fibonacci(7) = 13
>>>Calculated fibonacci(8) = 21
>>>Calculated fibonacci(9) = 34
>>>Calculated fibonacci(10) = 55
>>>55 CacheInfo(hits=8, misses=11, maxsize=4, currsize=4)
```

In these examples, we calculate several Fibonacci numbers with a cache capacity of four calculations. After calculating `fibonacci(10)`, the cache retains the seventh, eighth, ninth, and tenth numbers.

Consequently, `fibonacci(8)` can be retrieved without recalculation. However, when subsequently requesting `fibonacci(5)`, we discover this value has been evicted from the cache, necessitating a fresh calculation.

For most applications, cache constraints are unnecessary, and `@functools.cache` can be used directly.

```
print(fibonacci(20), fibonacci.cache_info())
print(fibonacci(5), fibonacci.cache_info())
# with default cache size which is 128
>>>6765 CacheInfo(hits=18, misses=21, maxsize=128, currsize=21)
>>>5 CacheInfo(hits=19, misses=21, maxsize=128, currsize=21)

#with cache size 4
CacheInfo(hits=18, misses=21, maxsize=4, currsize=4)
CacheInfo(hits=21, misses=27, maxsize=4, currsize=4)
```

18 Dependency Management

This section provides a comprehensive overview of two critical aspects of Python development: understanding the `__init__.py` file's role in Python packages and mastering pip for dependency management. These tools are essential for creating maintainable, shareable Python code and ensuring reproducibility across different environments.

18.1 The `__init__.py` File

The `__init__.py` file serves several crucial functions in Python packaging:

- It designates a directory as a Python package
- It initializes package attributes and submodules
- It controls what gets imported with wildcard imports
- It can implement package-level documentation
- It manages namespace packages (in Python 3.3+)

18.1.1 Basic Usage

The simplest `__init__.py` file can be completely empty. Even an empty file signals to Python that the directory should be treated as a package.

```
# This file can be empty
```

Here's how packages are typically structured, we will use this structure as an example:

```
src/                        # Top-level package
  __init__.py               # Makes mypackage a package
  moduleA.py                # A module in mypackage
  utils/                    # Subpackage
    __init__.py             # Makes subpackage a package
    moduleB.py              # A module in subpackage
```

```
#moduleA.py
class ClassA:
    def __init__(self):
        print("An Object of Class A is created!")
```

```
#moduleB.py
class ClassB:
    def __init__(self):
        print("An Object of Class B is created!")
```

18.1.2 Controlling Package Exports

The `__init__.py` file can be used to flatten package hierarchies, making imports more convenient:

```
from .moduleA import ClassA
from .utils.moduleB import ClassB

# Now users can do:
# from mypackage import ClassA, ClassB
# Instead of:
# from mypackage.moduleA import ClassA
# from mypackage.subpackage.moduleB import ClassB
```

The `__all__` list in `__init__.py` controls what is imported when a user writes `from package import *`:

```python
from .moduleA import ClassA
from .utils.moduleB import ClassB

__all__ = ['ClassA', 'ClassB']
```

Then, inside any file in the project, you can import all the items listed in `__all__`:

```python
from src import *

a = ClassA()
b = ClassB()
```

18.2 Pip Commands

```bash
# Install a package
pip install package_name

# Install a specific version
pip install package_name==1.2.3

# Upgrade a package
pip install --upgrade package_name

# Uninstall a package
pip uninstall package_name

# List installed packages
pip list

# Show detailed info about a package
pip show package_name
```

18.2.1 Working with Requirements Files

A `requirements.txt` file lists all dependencies:

```
# Comments are supported
numpy==1.24.3
pandas>=2.0.0,<3.0.0
matplotlib~=3.7.1  # Compatible release
requests==2.31.0
scipy  # No version specified, gets latest
```

- `==`: Exact version
- `>=`: Minimum version
- `<=`: Maximum version
- `>`, `<`: Greater than, less than
- `!=`: Not equal to
- `~=`: Compatible release (equivalent to `>=` current version `<` next major)

```bash
pip install -r requirements.txt
```

18.2.2 Generating Requirements

Capture your current environment:

```
pip freeze > requirements.txt
```

Instead of manually specifying exact versions, create a high-level `requirements.in` file using `pip-tools`:

```
pip install pip-tools
```

Creating a `requirements.in` File:

```
# High-level dependencies without exact versions
numpy
pandas>=2.0.0
matplotlib
requests
```

Compiling with pip-compile to generate a pinned `requirements.txt`:

```
pip-compile requirements.in
```

This creates a `requirements.txt` with exact versions of all dependencies, including transitive dependencies.

You can use pip to ensure your environment exactly matches requirements:

```
pip-sync requirements.txt
```

Unlike `pip install -r`, `pip-sync` will uninstall packages not in the requirements file.

19 Log Files

Python's built-in logging system helps developers track crucial information about program execution. The standard `logging` module enables you to capture execution details, error states, and usage metrics without additional packages. This system allows you to define loggers, customize severity levels, format message output, and save logs to files for future analysis.

- Logging is the practice of recording program execution data for later review
- Effective logs support debugging, performance optimization, and usage monitoring
- Python logging implementation involves configuring logger objects and appropriate severity thresholds
- Using dedicated logging libraries provides structure and output control
- Logging offers significant advantages over print statements, reducing maintenance overhead and enabling selective output filtering

19.1 Logging Module

To understand how the `logging` module functions, try this example in the Python standard REPL:

```
import logging
logging.warning("A Warning msg!")

>>>WARNING:root:A Warning msg!
```

The output displays the severity level followed by `root` (the default logger's name) and then your message. This represents the default format, which can be customized to include timestamps and additional information.

In this example, you're utilizing the root logger with a `WARNING` level message. Log levels constitute a fundamental aspect of logging, with five standard levels indicating event severity. Each level has a corresponding function for logging events at that severity.

Log Level	Function	Description
DEBUG	`logging.debug()`	Provides detailed information valuable for developers.
INFO	`logging.info()`	Offers general information about program execution.
WARNING	`logging.warning()`	Indicates something that requires attention.
ERROR	`logging.error()`	Alerts about unexpected problems in your program.
CRITICAL	`logging.critical()`	Reports serious errors that may have crashed your application.

The `logging` module provides a default logger that enables immediate logging without extensive configuration. However, observe the following behavior:

```
logging.debug("This is a debug message")
logging.info("This is an info message")
logging.warning("This is a warning message")
logging.error("This is an error message")
logging.critical("This is a critical message")

>>>WARNING:root:This is a warning message
>>>ERROR:root:This is an error message
>>>CRITICAL:root:This is a critical message
```

Notice how the `debug()` and `info()` messages don't appear. By default, the `logging` module only displays messages with `WARNING` severity or higher. You can modify this behavior by configuring the logging module to capture events of all levels.

19.2 Adjusting the Log Level and Formatting the Output

The `logging` module provides a `basicConfig()` function for initial setup and log level adjustment. To configure the root logger's level:

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug("This will get logged.")


>>>DEBUG:root:This will get logged.
```

The `level` parameter determines which message severities are recorded. You can specify levels using constants, numeric values, or string equivalents:

Constant	Numeric Value	String Value
`logging.DEBUG`	10	`DEBUG`
`logging.INFO`	20	`INFO`
`logging.WARNING`	30	`WARNING`
`logging.ERROR`	40	`ERROR`
`logging.CRITICAL`	50	`CRITICAL`

When you set a specific log level, all messages at that level and higher will be recorded.

By default, logs include the level, logger name, and message. You can enhance logs with additional data using the `format` parameter,the format parameter accepts a string that can contain a number of predefined attributes:

Attribute name	Format	Description
args	*Not formatted directly*	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the `LogRecord` was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the `LogRecord` was created (as returned by `time.time_ns()` / 1e9).
exc_info	*Not formatted directly*	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when `Formatter.format()` is invoked.
module	`%(module)s`	Module (name portion of `filename`).
msecs	`%(msecs)d`	Millisecond portion of the time when the `LogRecord` was created.
msg	*Not formatted directly*	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object.
name	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).

```
import logging
logging.basicConfig(format="%(levelname)s:%(name)s:%(message)s")
logging.warning("Hello, Warning!")

>>>WARNING:root:Hello, Warning!
```

This uses printf-style formatting. Alternative styles include string templates (`${}`) and modern Python formatting with curly braces (`{}`). Specify the style with the `style` parameter:

```
import logging
logging.basicConfig(format="{levelname}:{name}:{message}", style="{") # style is % by default
logging.warning("Hello, Warning!")

>>>WARNING:root:Hello, Warning!
```

Note: `basicConfig()` only works if the root logger hasn't been configured previously. Any logging function automatically calls `basicConfig()` without arguments if it hasn't been called yet. So, for example, once you call logging.info(), you'll no longer be able to configure the root logger with basicConfig().

```
import logging
logging.info("Call before basckConfig!") # remove this one you will see :: after levelname
logging.basicConfig(format="%(levelname)s::%(name)s:%(message)s", style="%") # double :: after levelname
logging.warning("Hello, Warning!")

>>WARNING:root:Hello, Warning! # output is single :, which shows you cannot configure it again once the root logger is called!
```

Timestamps help track when events occur, useful for performance monitoring and error pattern detection. By default, `asctime` also shows you milliseconds. If you don't need to be that exact or if you want to customize the timestamp, then you must add datefmt to your baseConfig() call::

```
import logging
logging.basicConfig(
    format="{asctime} - {levelname} - {message}",
    style="{",
    datefmt="%Y-%m-%d %H:%M:%S",)

logging.error("Something went wrong!")

>>>2025-04-21 11:07:10 - ERROR - Something went wrong!
```

The `datefmt` parameter controls timestamp formatting using directives like year (`%Y`), month (`%m`), day (`%d`), hour (`%H`), minutes (`%M`), and seconds (`%S`). For a complete list of time directives, refer to the `time.strftime()` documentation.

19.3 Logging to A File

While console logging is useful during development, archiving logs in files allows for long-term storage and analysis. To direct logs to a file, use the `filename` parameter in `basicConfig()`:

```
import logging
logging.basicConfig(
    filename="app.log",
    encoding="utf-8", #UTF-8 is a character encoding standard that represents each character in a document as a specific sequence o
    filemode="a",
    format="{asctime} - {levelname} - {message}",
    style="{",
   datefmt="%Y-%m-%d %H:%M",
)

logging.warning("Save to app.log!")
```

This configuration saves logs to `app.log` instead of displaying them in the console. The `filemode="a"` parameter ensures new logs are appended rather than overwriting existing content.

The resulting log file contains plain text entries:

```
2025-04-21 11:29 - WARNING - Save to app.log!
```

For advanced logging architectures, consider organizing logs in date-formatted folders or structuring them as CSV files for easier parsing and analysis. We will have a comprehensive example at the end of this section.

Most log messages need to include runtime information. Python's f-strings provide a convenient way to include variable data:

```python
import logging
logging.basicConfig(
     format="%(asctime)s - %(levelname)s - %(message)s",
     style="%",
     datefmt="%Y-%m-%d %H:%M",
    level=logging.DEBUG,
)

info = "an example"
logging.debug("info={}".format(info))
```

The logging system can record full exception stack traces by setting `exc_info=True`:

```python
import logging
logging.basicConfig(
    filename="app.log",
    encoding="utf-8",
    filemode="a",
    format="{asctime} - {levelname} - {message}",
    style="{",
    datefmt="%Y-%m-%d %H:%M",
)

numerator = 5
denominator = 0
try:
    res = numerator / denominator
except ZeroDivisionError:
    logging.error("ZeroDivisionError", exc_info=True)
```

The resulting log entry includes the complete stack trace:

```
2025-04-21 11:37 - ERROR - ZeroDivisionError
Traceback (most recent call last):
  File "/Users/dengq/Documents/CSC111-2025Spring/project/test.py", line 14, in <module>
    res = numerator / denominator
ZeroDivisionError: division by zero
```

For convenience, the `logging.exception()` function automatically includes exception information:

```python
import logging
logging.basicConfig(
    filename="app.log",
    encoding="utf-8",
    filemode="a",
    format="{asctime} - {levelname} - {message}",
    style="{",
    datefmt="%Y-%m-%d %H:%M",
)

numerator = 5
denominator = 0
try:
    res = numerator / denominator
```

```
except ZeroDivisionError:
    logging.exception("DonutCalculationError") # you do not need to set exc_info=True manually
```

This is equivalent to `logging.error(exc_info=True)` and should only be used within exception handlers. While `logging.exception()` always uses the `ERROR` level, you can use any logging function with `exc_info=True` to capture exceptions at different severity levels.

19.4 Custom Logger

You can instantiate a custom logger by calling the `logging.getLogger()` function:

```
import logging
logger = logging.getLogger(__name__)
logger.warning("Newly created logger!")


>>>Newly created logger!
```

Although any string identifier would work as the logger name, using `__name__` is considered best practice. **This approach automatically uses the module's name in the Python package namespace, creating a natural hierarchy of loggers that reflects your application's structure.**

Notice that unlike the root logger, the output lacks formatting information such as the logger's name or log level. Custom loggers cannot be configured with `basicConfig()`. Instead, they require handlers and formatters for more flexible configuration.

19.4.1 more about `__name__`

In Python, `__name__` is a built-in variable that represents the name of the current module. It is automatically set by the Python interpreter when a module is loaded.

`__name__` exhibits two primary behaviors:

- When a module is run directly as the main program, `__name__` is set to the string `"__main__"`
- When a module is imported into another module, `__name__` is set to the module's filename without the `.py` extension

The most common usage pattern for `__name__` is the module execution guard:

```
if __name__ == "__main__":
    # Code here will only run when the file is executed directly
    # Will not run when the file is imported as a module
```

Consider a file named `example.py`:

```
def some_function():
    return "This function was called"

print(f"The value of __name__ is: {__name__}")

if __name__ == "__main__":
    print("This file is being run directly")
    print(some_function())
else:
    print("This file is being imported as a module")
```

When run directly (`python example.py`):

```
The value of __name__ is: __main__
This file is being run directly
This function was called
```

When imported in another file:

```
The value of __name__ is: example
This file is being imported as a module
```

This pattern allows you to:

- Use the file as a standalone script for testing
- Import and use its functions in other scripts without running the tests

19.4.2 Handlers

Handlers direct log messages to their destinations. This architecture allows a single logging event to be processed by multiple handlers, each with its own configuration, that means you can send your logs to multiple places when they're generated.

```
import logging
logger = logging.getLogger(__name__)
console_handler = logging.StreamHandler()
file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
```

In this example:

- `StreamHandler` sends logs to the console
- `FileHandler` writes logs to a file, configured with path, mode, and encoding

After instantiating handlers, attach them to your logger with the `addHandler()` method:

```
import logging
logger = logging.getLogger(__name__)
console_handler = logging.StreamHandler()
file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
logger.addHandler(console_handler)
logger.addHandler(file_handler)
print(logger.handlers)

>>>[<StreamHandler <stderr> (NOTSET)>, <FileHandler /Users/dengq/Documents/CSC111-2025Spring/project/app.log (NOTSET)>]
```

The `handlers` property lists all attached handlers. Each representation shows:

- For the StreamHandler, the logs are written to the standard error stream (stderr) stream, which is the output channel that Python uses by default.
- For the FileHandler, you can see the location where your log records will be saved.
- In the parentheses of the class representation, you'll see the log level of your handlers. Currently, the log level is NOTSET. As you might expect, NOTSET means that the log level for the logging handlers is not set, yet.

When you log a message, both handlers process it:

```
logger.warning("A warning msg from my costom logger!")
```

The `StreamHandler` immediately displays the message in the console while `FileHandler` simultaneously writes it to `app.log`.

19.4.3 Formatters

While handlers determine the destination of log messages, formatters control their presentation. Formatters allow you to structure log records with metadata such as timestamps, log levels, and other contextual information.

To implement formatters, you must instantiate the `Formatter` class and attach it to a handler:

```
import logging
logger = logging.getLogger(__name__)
console_handler = logging.StreamHandler()
file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
logger.addHandler(console_handler)
logger.addHandler(file_handler)
formatter = logging.Formatter(
   "{asctime} - {levelname} - {message}",
    style="{",
    datefmt="%Y-%m-%d %H:%M",
)

console_handler.setFormatter(formatter)
logger.warning("A warning msg from my costom logger!")
```

The formatter instance defines how log messages will appear when processed by a particular handler. In this example, a formatter that displays timestamp, log level, and message is created and attached to `console_handler` using the `setFormatter()` method.

Note: Unlike `addHandler()`, which is a method of the `Logger` class, `setFormatter()` is a method of the `Handler` class.

When a logging event occurs, each handler processes it according to its configuration. In the example above, although both handlers receive the warning message, only the console output is formatted because `file_handler` has no formatter attached. This flexibility allows different presentation formats across output destinations.

19.4.4 Configuring Log Levels for Custom Loggers

Log levels provide control over which messages are processed. With custom loggers, you can configure different severity thresholds for various handlers.

When first created, custom loggers have their own default settings:

```
import logging
logger = logging.getLogger(__name__)
print(logger.level) # 0
print(logger) # <Logger __main__ (WARNING)>
print(logger.parent) # <RootLogger root (WARNING)>
```

A new logger's `level` property is `0` (representing `NOTSET`), yet its string representation shows `WARNING`. This occurs because custom loggers inherit their effective log level from their parent logger when not explicitly configured.

The `getEffectiveLevel()` method reveals the actual threshold being applied:

```
print(logger.getEffectiveLevel()) # 30 which means logging.WARNING
```

You can configure a logger's level using either numeric values or string identifiers:

```
import logging
logger = logging.getLogger(__name__)
logger.setLevel(10)
print(logger) # <Logger __main__ (DEBUG)>
logger.setLevel("INFO")
print(logger) # <Logger __main__ (INFO)>
```

When handlers are added to a logger, they respect the logger's level threshold:

```python
import logging
logger = logging.getLogger(__name__)
logger.setLevel("INFO")
formatter = logging.Formatter("{levelname} - {message}", style="{")
console_handler = logging.StreamHandler()
console_handler.setFormatter(formatter)
logger.addHandler(console_handler)
logger.debug("A debug level info.") # will not be printed
logger.info("An info level msg.") # INFO - An info level msg.
```

An important principle to understand is that handlers cannot process messages below their logger's level, even if the handler itself is configured for lower levels:

```python
import logging
logger = logging.getLogger(__name__)
logger.setLevel("INFO")
formatter = logging.Formatter("{levelname} - {message}", style="{")
console_handler = logging.StreamHandler()
console_handler.setFormatter(formatter)
logger.addHandler(console_handler)
console_handler.setLevel("DEBUG") # lower than logger's level
logger.debug("A debug level info.") # Will not be printed
```

Despite configuring the handler for `DEBUG` level, the debug message doesn't appear because the logger's `INFO` threshold filters it before it reaches any handlers.

Note: The logger establishes the minimum threshold; handlers can only further restrict which messages are processed, not broaden the scope.

A common development pattern sets the logger to the lowest needed level while configuring handlers with different thresholds:

```python
import logging
logger = logging.getLogger(__name__)
logger.setLevel("DEBUG")
formatter = logging.Formatter("{levelname} - {message}", style="{")

console_handler = logging.StreamHandler()
console_handler.setLevel("DEBUG")
console_handler.setFormatter(formatter)
logger.addHandler(console_handler)

file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
file_handler.setLevel("WARNING")
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

logger.debug("A debug info!")
logger.warning("A warning!")
logger.error("An error!")
```

In this configuration, the console shows all messages while the log file contains only warnings and errors.

19.4.5 Filters

While log levels provide broad control over message processing, some scenarios require more precise filtering. For instance, you might want to isolate messages of a specific level rather than processing all messages above a certain threshold.

The `Filter` mechanism allows for fine-grained control over which log records a handler processes. According to the logging specification, a filter can be implemented in three ways:

1. A **subclass** of `logging.Filter()` with an overridden `filter()` method
2. A **class** containing a `filter()` method
3. A **callable** that functions like a `filter()` method

For subclasses and classes, the `filter()` method must accept a log record parameter and return a Boolean value. The method typically includes a conditional statement that evaluates the provided record.

For callables, a simple function with a parameter for the log record is sufficient. The return value must be Boolean, determining whether the handler processes the record.

The callable approach offers a straightforward implementation for basic filtering requirements:

```
import logging
def show_only_debug(record):
    return record.levelname == "DEBUG"


logger = logging.getLogger(__name__)
logger.setLevel("DEBUG")
formatter = logging.Formatter("{levelname} - {message}", style="{")

console_handler = logging.StreamHandler()
console_handler.setLevel("DEBUG")
console_handler.setFormatter(formatter)
console_handler.addFilter(show_only_debug)
logger.addHandler(console_handler)

file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
file_handler.setLevel("WARNING")
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

logger.debug("A debug info!") # will be printed to console

logger.warning("A warning info!") # # will not be printed to console
logger.error("An error!") # # will not be printed to console
```

In this example:

- The `show_only_debug()` function returns `True` only when a log record's `levelname` attribute equals `"DEBUG"`
- The function is attached to `console_handler` via the `addFilter()` method
- Although `console_handler` is configured to process all messages at `DEBUG` level and above, the filter restricts output to only `DEBUG` messages
- `file_handler` has no filter, so it continues to process all messages at `WARNING` level and above

Without the filter, `console_handler` would display all three test messages. With the filter in place, only the debug message appears in the console, while the warning and error messages are silently filtered out but still written to the log file.

19.4.6 Practical Built-in Handlers

The `RotatingFileHandler` creates new log files when the current file reaches a specified size limit. This is useful for applications that generate logs continuously and need to keep log file sizes manageable.

```
import logging
from logging.handlers import RotatingFileHandler

# Create a logger
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create a rotating file handler
# Parameters:
```

```
# - filename: name of the log file
# - maxBytes: maximum size in bytes before rotation
# - backupCount: number of backup files to keep
handler = RotatingFileHandler(
    'app.log',
    maxBytes=10485760,  # 10MB
    backupCount=5
)

handler.setLevel(logging.INFO)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)

# Log messages
logger.info('This is an info message')
```

When a log file reaches the size specified by `maxBytes`, the following happens:

1. The current log file (`app.log`) is renamed to `app.log.1`
2. If `app.log.1` already exists, it is renamed to `app.log.2`
3. This continues until we reach `backupCount`
4. If `app.log.backupCount` already exists, it is deleted
5. A new empty `app.log` file is created

The `TimedRotatingFileHandler` creates new log files based on time intervals rather than file size. This is ideal for applications where logs should be organized by time periods (e.g., daily, weekly, or monthly logs).

```
import logging
from logging.handlers import TimedRotatingFileHandler

# Create a logger
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create a timed rotating file handler
# Parameters:
# - filename: name of the log file
# - when: time interval type ('S', 'M', 'H', 'D', 'W0'-'W6', 'midnight')
# - interval: number of time units (default is 1)
# - backupCount: number of backup files to keep
handler = TimedRotatingFileHandler(
    'timed_app.log',
    when='D',           # Daily rotation
    interval=1,         # Every day
    backupCount=30      # Keep logs for 30 days
)

handler.setLevel(logging.INFO)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)

# Log messages
logger.info('This is a timed rotation log message')
```

The `when` parameter accepts several values:

- `'S'`: Seconds
- `'M'`: Minutes
- `'H'`: Hours

- `'D'`: Days
- `'W0'` - `'W6'`: Weekdays (0=Monday, 6=Sunday)
- `'midnight'`: Roll over at midnight

19.5 Save logs in CSV Files

19.5.1 CSVFormatter Class

```
import logging
import csv
from datetime import datetime

class CSVFormatter(logging.Formatter):
    def __init__(self):
        super().__init__()
        self.fields = ['timestamp', 'level', 'message', 'module', 'function', 'line']

    # override format function
    def format(self, record):
        # Create a dictionary with the log information
        log_data = {
            'timestamp': datetime.fromtimestamp(record.created).strftime('%Y-%m-%d %H:%M:%S'),
            'level': record.levelname,
            'message': record.getMessage(),
            'module': record.module,
            'function': record.funcName,
            'line': record.lineno
        }
        return log_data

    @property
    def header(self):
        return self.fields
```

The `CSVFormatter` class extends Python's standard `logging.Formatter` class with the following features:

- **Initialization**: Defines the fields that will be included in the CSV file.
- **format() method**: Overrides the parent class's format method to:
  - Accept a log record object
  - Extract relevant information from the record
  - Convert the timestamp to a human-readable format
  - Return a dictionary where keys match the defined fields
- **header property**: Provides access to the field names for creating the CSV header

Note that instead of returning a formatted string as most formatters do, this formatter returns a dictionary that will be directly used by the CSV writer.

19.5.2 CSVHandler Class

```
class CSVHandler(logging.Handler):
    def __init__(self, filename):
        super().__init__()
        self.filename = filename
        self.formatter = CSVFormatter()

        # Write CSV header
        with open(self.filename, 'w', newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=self.formatter.header)
            writer.writeheader()
```

```
    # override emit function
    def emit(self, record):
        log_entry = self.formatter.format(record)
        with open(self.filename, 'a', newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=self.formatter.header)
            writer.writerow(log_entry)
```

The `CSVHandler` class extends the base `logging.Handler` class:

- **Initialization**:
  - Accepts a filename parameter
  - Creates a `CSVFormatter` instance
  - Opens the CSV file in write mode ('w'), which will overwrite any existing file
  - Writes the header row using the field names from the formatter
- **emit() method**: Called whenever a log record needs to be processed
  - Formats the record using the `CSVFormatter`
  - Opens the file in append mode ('a')
  - Uses `csv.DictWriter` to write the log entry as a new row

Key implementation details:

- The `newline=''` parameter ensures consistent line endings across platforms
- File handling is done inside context managers (with statements) to ensure proper file closure
- The file is opened in append mode for each log entry, which is less efficient but more robust against data loss if the program crashes

19.5.3 An example

```
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Add the CSV handler
handler = CSVHandler("application_logs.csv")
logger.addHandler(handler)

logger.debug("This is a debug message")
logger.info("Application started")

# Simulate an error
try:
    result = 10 / 0
except Exception as e:
    logger.error(f"Error occurred: {str(e)}")

logger.info("Process completed")
```

This section demonstrates the logger in action:

- Initializes the logger with the setup function
- Logs messages at different levels (DEBUG, INFO)
- Demonstrates error logging within an exception handler
- Shows that logging continues after error handling

The division by zero will raise a `ZeroDivisionError`, which is caught and logged at the ERROR level. When run, this code generates a CSV file named "application_logs.csv" with the following structure:

```
timestamp,level,message,module,function,line
2025-04-21 13:15:04,DEBUG,This is a debug message,test,<module>,53
2025-04-21 13:15:04,INFO,Application started,test,<module>,54
2025-04-21 13:15:04,ERROR,Error occurred: division by zero,test,<module>,60
2025-04-21 13:15:04,INFO,Process completed,test,<module>,62
```

Each row represents a log entry with consistent fields that can be easily analyzed using spreadsheet software or data analysis tools.

The current handler doesn't support log rotation. This could be implemented by extending `RotatingFileHandler` instead:

```python
from logging.handlers import RotatingFileHandler
class CSVHandler(RotatingFileHandler):
    def __init__(self, filename, maxBytes=0, backupCount=0):
        super().__init__(filename, maxBytes=maxBytes, backupCount=backupCount)
        self.formatter = CSVFormatter()

        # Write header to the initial file
        with open(self.baseFilename, 'w', newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=self.formatter.header)
            writer.writeheader()

    def emit(self, record):
        if self.shouldRollover(record):
            self.doRollover()
            # Write header to the new file
            with open(self.baseFilename, 'w', newline='') as csvfile:
                writer = csv.DictWriter(csvfile, fieldnames=self.formatter.header)
                writer.writeheader()

        log_entry = self.formatter.format(record)
        with open(self.baseFilename, 'a', newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=self.formatter.header)
            writer.writerow(log_entry)
```

I believe you know how to write a Timed Rotating CSVFileHandler by yourself.