2. Types

Good Note:

Python is a **dynamically typed** language, meaning you don't declare the type of a variable; the **type is inferred at runtime** based on the value assigned. Python has core built-in types, including: **int** (integers), **float** (floating-point numbers), **bool** (Boolean `True`/`False`), and **str** (strings). The type of any object can be checked using the built-in function **type()**.

One Example:

python

```
x = 10          # x is inferred as int
y = 3.14        # y is inferred as
float
is_valid = True # is_valid is
inferred as bool
name = "Python" # name is inferred as
str

print(type(x))
print(type(y))

# Output:
# <class 'int'>
# <class 'float'>
```

1. Constants

Good Note:

Python **lacks true, enforced constants**. A "constant" in Python is just a **variable** whose value is **not intended to be changed** after assignment. This intention is communicated using a **naming convention**: the identifier is written in **ALL_CAPS** with words separated by underscores (e.g., `MAX_SIZE`). Since the language doesn't prevent modification, the concept relies entirely on **programmer discipline** for immutability.

One Example:

python

```
# The standard convention for
defining a constant
PI = 3.14159
SPEED_OF_LIGHT = 299792458  # Meters
per second

# Technically, Python allows you to
reassign it, but you should NOT
# PI = 3.0   # This is poor practice
and violates the convention
```

3. Identifiers

Good Note:

Identifiers are the **names** given to entities like variables, functions, classes, modules, etc. in Python. They must adhere to three strict rules:

1. Must start with a **letter** (`a-z`, `A-Z`) or an **underscore** (`_`).
2. Cannot contain **spaces** or special characters (like `!`, `@`, `#`, `%`).
3. Are **case-sensitive** (`MyVar` is different from `myvar`).
   Additionally, identifiers **cannot** be a reserved Python **keyword** (e.g., `for`, `if`, `while`).

One Example:

python

```
# Valid Identifiers:
_count = 0
Total_Sum = 100
user_id_2025 = "U123"

# Invalid Identifiers (will cause a
SyntaxError):
# 2nd_name = "Alice"  # Cannot start
with a digit
# total-score = 95    # Hyphens are
not allowed
# for = True          # Cannot use a
reserved keyword
```

4. Keywords

Good Note:

Keywords are a set of **reserved words** that have special meaning and functionality in Python. They are **case-sensitive** (e.g., `if` is a keyword, but `IF` is not) and **cannot be used** as identifiers (variable names, function names, etc.). They are used to define the syntax and structure of the Python language, controlling program flow (`if`, `for`, `while`), defining structures (`def`, `class`), and handling errors (`try`, `except`).

One Example:

python

```
# A simple example using three
essential keywords:
# 'def' defines a function
# 'if' controls selection
# 'return' defines the output value

def check_value(x):
    # The keyword 'if'
    if x > 10:
        # The keyword 'return'
        return True
    else:
        # The keyword 'else'
        return False

# You cannot use keywords as variable
names:
# while = 5       # This would cause
a SyntaxError
```

5. Comments

Good Note:

Comments are non-executable text added to code to improve **readability** and explain **what** the code does or **why** it was written. Python ignores them during execution. Single-line comments start with the **hash symbol (#)**. Python does not have a formal multi-line comment syntax; instead, programmers often use **triple quotes** (`"""..."""` or `'''...'''`) which, if not assigned to a variable, act as multi-line comments (often used for docstrings).

One Example:

python

```
# This is a single-line comment. It explains the next line.
age = 25  # Inline comment: Assigning the user's age

"""
This block of text is a string literal,
but is used here as a multi-line comment
because it is not assigned to any variable.
"""
result = age * 2
```

7. input()

Good Note:

The built-in `input()` function is used to prompt the user and **read a line of text** from the standard input (usually the keyboard). Crucially, the function **always returns the input as a string (str)**, even if the user types only digits. If numerical input is required (for calculations), you **must explicitly cast** the result using functions like `int()` or `float()`. The function accepts an optional argument (a string) which serves as the prompt displayed to the user.

One Example:

python

```
# The prompt "Enter your age: " is
displayed
age_str = input("Enter your age: ")
print(f"Type of input:
{type(age_str)}")

# Casting is required to use 'age' as
a number
age_int = int(age_str)
next_year_age = age_int + 1

# Output (assuming user enters 25):
# Enter your age: 25
# Type of input: <class 'str'>
# 26
```

6. print()

Good Note:

The built-in `print()` function is used for standard output. It is crucial to understand its behavior regarding arguments and separators.

1. **Function Introduction:** It automatically converts all arguments into strings and displays them on the console.
2. **end Keyword:** By default, `end` is set to the newline character (`\n`), meaning that after printing, the cursor moves to the next line. You can change this to keep subsequent output on the same line (e.g., `end=" "`, `end="..."`).
3. **sep Keyword:** This defines the separator placed **between multiple arguments** passed to the function. The default value is a **single space** (`" "`).
4. **Crucial Nuance:** `sep` is **ignored** if the function receives only one argument. If you use the string concatenation operator (+) to join variables before calling `print()`, the function receives a single string, and `sep` will not be used.

One Example:

python

```
A = 5
B = 10

# 1. Shows default 'sep' (space) and
default 'end' (\n)
print("Result:", A, B)
# Output: Result: 5 10

# 2. Shows custom 'end' and custom
'sep'
print(A, B, sep=" + ", end=" = ")
print(A + B)
# Output: 5 + 10 = 15
```

8. help() function

Good Note:

The built-in **help()** function launches Python's interactive help utility. It is used to get **documentation and detailed information** about modules, keywords, functions, classes, and methods directly within the interpreter. When called without arguments (`help()`), it enters an interactive mode. When called with an object as an argument (e.g., `help(str)` or `help(print)`), it displays the corresponding docstring (documentation) and signature, which is highly useful for quick reference while coding.

One Example:

python

```
# Launch help for the built-in
'range' function
help(range)

# Output (Snippet):
# Help on class range in module
builtins:
# class range(object)
#  |  range(stop) -> range object
#  |  range(start, stop[, step]) ->
range object
#  |
#  |  Return an object that produces
a sequence of integers
#  |  from start (inclusive) to stop
(exclusive) by step.
#  ...
```

9. PEMDAS (Operator Precedence and Associativity)

Good Note:

PEMDAS (or BODMAS/BEMDAS) defines the **order of operations** in arithmetic expressions. In Python, this hierarchy is known as **Operator Precedence**. Operations with higher precedence are evaluated first. If two operators have the same precedence (e.g., * and /), **Associativity** determines the order (usually left-to-right, though the exponentiation operator ** is right-to-left). **Parentheses () always have the highest precedence** and force the enclosed expression to be evaluated first.

One Example:

python

```
# Order: Exponent (**) > Multiply (*)
> Subtract (-)
result = 10 - 2 * 3 ** 2
# Step 1: 3 ** 2 = 9
# Step 2: 2 * 9 = 18
# Step 3: 10 - 18 = -8
print(result)

# Using Parentheses to force a
different order
result_with_paren = (10 - 2) * 3 ** 2
# Step 1: (10 - 2) = 8
# Step 2: 3 ** 2 = 9
# Step 3: 8 * 9 = 72
print(result_with_paren)
```

10. Binary Number System

Good Note:

The binary system is the base-2 system, using only **0s and 1s**. Python represents a binary literal with the prefix **0b** (e.g., 0b1011). The built-in function **bin()** converts an integer into its binary representation.

1. **Prefix:** The resulting string is prefixed with 0b (e.g., 0b1101).
2. **Datatype:** The `bin()` function **returns a string** (<class 'str'>), **not an integer**.

One Example:

python

```
# The prefix 0b tells Python this is
a binary literal (value is 13)
decimal_from_binary = 0b1101

# The bin() function converts the
decimal integer 15 to a string
binary_str = bin(15)

print(binary_str)
print(type(binary_str))

# Output:
# 0b1111
# <class 'str'>
```

12. Bitwise Operators

Here note starts:
Bitwise operators perform operations directly on the individual **bits** (0s and 1s) of integers. They are typically used for high-performance calculations, manipulating binary flags, and encryption. The operators are:

1. **AND (&):** Sets the bit to 1 if **both** bits are 1.
2. **OR (|):** Sets the bit to 1 if **at least one** bit is 1.
3. **XOR (^):** Sets the bit to 1 if the bits are **different**.
4. **NOT (~):** Flips the bits (inverts 0s to 1s and vice versa). This is equivalent to $-(\mathbf{x}+\mathbf{1})$ (e.g., ~10 is -11).

Example:

python

```
A = 10  # Binary: 0b1010
B = 4   # Binary: 0b0100

# Bitwise AND
print(A & B)  # 0b1010 & 0b0100 =
0b0000 -> Output: 0

# Bitwise OR
print(A | B)  # 0b1010 | 0b0100 =
0b1110 -> Output: 14

# Bitwise NOT
print(~A)     # ~10 = -(10 + 1) ->
Output: -11
```

11. bin() function

Note:

The built-in `bin()` function converts a decimal **integer** into its **binary string representation**.

1. **Input:** Requires an argument of type **int**.
2. **Output Type:** Always returns a **string** (<class 'str'>).
3. **Prefix:** The output string is always prefixed with **0b** (e.g., 0b1010) to identify it as a binary literal.
4. **Error:** Using a **float** as an argument (e.g., `bin(3.5)`) results in a **TypeError**.

Example:

python

```
num = 10
result_str = bin(num)

print(result_str)
print(type(result_str))

# Attempting binary literal
conversion from string fails
# print(int(result_str, 2)) # Use
int(s, base) to convert back

# Output:
# 0b1010
# <class 'str'>
```

13. Bit Masking

Bit masking is the technique of using bitwise operators with a specially crafted integer (the **mask**) to easily perform actions like setting, clearing, or checking the state of specific bits within another integer (the **data**).

1. **Checking/Extracting a Bit:** Use the **AND (&)** operator. A mask with a 1 at the desired position will isolate that bit. Example: `(Data & Mask)`.
2. **Setting a Bit (to 1):** Use the **OR (|)** operator. A mask with a 1 at the desired position will ensure that bit is set to 1, regardless of its current state. Example: `(Data | Mask)`.
3. **Clearing a Bit (to 0):** Use the **AND (&)** operator with the **NOT (~)** of the mask. The `~Mask` creates a mask with a 0 at the target position, forcing the bit to 0. Example: `(Data & ~Mask)`.
4. **Creating a Mask:** A mask is typically created using the left shift operator (<<). To target the n-th bit, the mask is `1 << n`.

Example:

python

4. **Creating a Mask:** A mask is typically created using the left shift operator (<<). To target the n-th bit, the mask is 1 << n.

Example:

python

```
# Assume Data is 5 (0b0101). We want
to check the 2nd bit (index 2).
DATA = 5
MASK = 1 << 2  # MASK is 4 (0b0100)

# Checking the bit
is_set = (DATA & MASK)
print(is_set)   # Output: 4 (True,
since result is non-zero)

# Setting the 0th bit (index 0)
SET_MASK = 1 << 0  # SET_MASK is 1
(0b0001)
new_data = DATA | SET_MASK
print(new_data) # Output: 5 (0b0101 |
0b0001 = 0b0101, bit already set)

# Clearing the 2nd bit
CLEARED_DATA = DATA & ~MASK
print(CLEARED_DATA) # Output:
1 (0b0101 & ~0b0100 = 0b0101 &
0b1011... = 0b0001)
```

14. Relational and Equality Operators

These operators are used to **compare** two values and always return a **Boolean** result (True or False).

1. **Relational Operators:** Used for magnitude comparison:
   - Greater than: >
   - Less than: <
   - Greater than or equal to: >=
   - Less than or equal to: <=
2. **Equality Operators:**
   - Equal to: == (Checks if **values** are the same)
   - Not equal to: != (Checks if **values** are different)
3. **Chaining:** Python allows mathematical-style **chained comparisons**, which are evaluated implicitly using the and operator. Example: $\mathbf{a < b < c}$ is the same as $\mathbf{a < b}$ and $\mathbf{b < c}$.
4. **String Comparison:** Strings are compared lexicographically (dictionary order) based on the Unicode values of their characters. Uppercase letters come before lowercase letters (`'A' < 'a'` is True).

Example:

python

```
x = 10
y = 20

# Relational comparison
print(x < y)      # Output: True

# Chained comparison
print(10 <= x < 15)  # 10 <= 10 AND
10 < 15. Output: True

# String comparison (lexicographical)
print("apple" < "Apple") # Output:
False (a > A)
```

14. Relational and Equality Operators

These operators are used to **compare** two values and always return a **Boolean** result (True or False).

1. **Relational Operators:** Used for magnitude comparison:
   - Greater than: >
   - Less than: <
   - Greater than or equal to: >=
   - Less than or equal to: <=
2. **Equality Operators:**
   - Equal to: == (Checks if **values** are the same)
   - Not equal to: != (Checks if **values** are different)
3. **Chaining:** Python allows mathematical-style **chained comparisons**, which are evaluated implicitly using the and operator. Example: $\mathbf{a < b < c}$ is the same as $\mathbf{a < b}$ and $\mathbf{b < c}$.
4. **String Comparison:** Strings are compared lexicographically (dictionary order) based on the Unicode values of their characters. Uppercase letters come before lowercase letters (`'A' < 'a'` is True).

Example:

python

```
x = 10
y = 20

# Relational comparison
print(x < y)      # Output: True

# Chained comparison
print(10 <= x < 15)  # 10 <= 10 AND
10 < 15. Output: True

# String comparison (lexicographical)
print("apple" < "Apple") # Output:
False (a > A)
```

15. not Operator

The logical **not** operator is a **unary** operator (takes only one operand) that inverts the Boolean state of its operand.

1. **Operation:** It returns `True` if the operand is considered **False** (or "falsy") and returns `False` if the operand is considered **True** (or "truthy").
2. **Truthiness/Falsiness:** In Python, values considered **False** (Falsy) include: `False`, `None`, the integer `0`, empty containers (empty string `""`, empty list `[]`, empty tuple `()`, empty dictionary `{}`), and empty set. Any other value is considered **True** (Truthy).
3. **Application:** It is often used to negate a condition within an `if` statement or a `while` loop.

Example:

python

```
is_adult = False
data = []
age = 18

# 1. Inverting a simple boolean
print(not is_adult)    # not False ->
Output: True

# 2. Testing falsiness of an empty
list
print(not data)        # not []
(which is Falsy) -> Output: True

# 3. Negating a relational expression
print(not (age < 21))  # not (True)
-> Output: False
```

16. Short-Circuit Evaluation (and, or)

Short-circuit evaluation means the Python interpreter stops evaluating a complex logical expression (`and` or `or`) as soon as the **final result is determined**.

1. **`and` Operator:** If the **left operand is Falsy** (e.g., `0`, `False`, `""`), it immediately returns that Falsy value without checking the right operand. If the left operand is Truthy, it returns the value of the right operand.
2. **`or` Operator:** If the **left operand is Truthy** (e.g., `1`, `True`, `"hello"`), it immediately returns that Truthy value without checking the right operand. If the left operand is Falsy, it returns the value of the right operand.
3. **Benefit:** This prevents errors (like division by zero) if the second condition is dependent on the first being true/false.

Example:

python

```
x = 0
y = 10

print(x and (10 / 0))
print(y or (10 / 0))

# Output:
# 0
# 10
```

17. Selection: if, elif, and else

Selection structures are used to execute specific blocks of code based on whether a condition evaluates to True or False.

1. **Structure:** `if` starts the block. `elif` (Else If) provides additional checks only if prior conditions were False. `else` is the optional catch-all block that executes if **all** conditions fail.
2. **Mutual Exclusion:** In the selection context, the entire `if/elif/else` structure executes **at most one** code block.
3. **Loop Nuance (MCA Detail):** The `else` keyword can also be attached to **for** and **while** loops. In this context, the else block executes **ONLY IF the loop completes normally** (i.e., the loop finishes all iterations or the while condition becomes False).
4. **Exception:** The loop's else block is **SKIPPED** if the loop is terminated prematurely by a **break** statement.
5. **Indentation:** Consistent indentation defines the code blocks for all structures.

Example:

python

```
score = 85

if score >= 90:
    print("Grade A")
elif score >= 80:
    print("Grade B")
else:
    print("Grade C")

# Output:
# Grade B
```

Mutual exclusion describes the property of an **if/elif/else** block where **only one** code block is executed, and all other possible paths are logically excluded.

1. **Selection Logic:** The conditions are checked sequentially. As soon as the interpreter finds a condition (`if` or `elif`) that evaluates to `True`, it executes that specific block and then **exits the entire structure**.
2. **Guaranteed Single Path:** This guarantees that even if a subsequent `elif` condition were also `True`, it will never be checked or executed once a preceding block has run.
3. **Use Case:** Ensures that complex conditions are handled with high efficiency and prevents contradictory results from being executed.

Example:

python

```
time = 14

if time < 12:
    print("Morning")
elif time < 18:
    print("Afternoon")
else:
    print("Evening")

# Output:
# Afternoon
```

19. Indentation (Space vs. Tab)

Indentation is fundamental in Python, as it defines **block structure** (scope) where other languages use braces (`{}`). This makes Python code visually clean but requires strict adherence.

1. **Block Definition:** After a statement ending with a colon (`:`)—like `if`, `def`, `for`, or `while`—the code belonging to that block **must** be indented.
2. **Consistency:** The number of spaces used for indentation **must be consistent** within a single block. The standard and recommended practice is **4 spaces.**
3. **Space vs. Tab (Crucial Detail):** Python strongly discourages mixing spaces and tabs. If you mix them, the interpreter will raise an **IndentationError** because a tab and an equivalent number of spaces (e.g., 4) are treated differently, leading to inconsistent block depths.
4. **Best Practice:** Always configure your editor to use **4 spaces** for indentation.

Example:

python

```
x = 10

if x > 5:
    print("Greater")
    print("Than 5")
else:
    print("Less")
    # This line has incorrect
indentation and would cause an error
# print("Than or equal")
```

20. Nesting

Nesting is the practice of placing one control structure (like a loop or a selection statement) **inside** the code block of another. This allows for complex, multi-layered logic, especially for handling multi-dimensional data or complex decision-making processes.

1. **Selection Nesting:** Placing an **`if/elif/else`** structure inside another **`if/elif/else`** block. The inner block is only evaluated if the condition of the outer block is met.
2. **Loop Nesting:** Placing a **`for`** or **`while`** loop inside another. The inner loop will execute completely for **every single iteration** of the outer loop.
3. **Cross-Nesting:** A loop can contain an **`if`** block, and an **`if`** block can contain a loop.
4. **Indentation:** Each level of nesting requires an additional level of **consistent indentation**.

Example:

python

```
x = 5

if x > 0:
    if x % 2 == 0:
        print("Positive and Even")
    else:
        print("Positive and Odd")

# Output:
# Positive and Odd
```

21. pass

The **pass** statement is a **null operation**; when executed, nothing happens. It acts as a **placeholder** in Python code where the syntax requires a statement but the programmer needs to defer writing the actual code block.

1. **Syntactic Necessity:** Used to satisfy structural rules (e.g., after an **if** condition, function **def**, or loop) that legally demand an indented block of code, even if that block is currently empty.
2. **Execution Flow (Deep Detail):** The `pass` statement **does not affect the execution of preceding or succeeding code** within the same indented block. Execution flows sequentially; `pass` is simply an instruction to "do nothing" at its specific position, and then proceed to the next line of code.
3. **Use Cases:** Defining interface-like classes or functions, or explicitly ignoring an exception in a `try...except` block.

Example:

python

```
x = 10

if x > 5:
    print("Pre-pass statement
executed")
    x = 20
    pass
    print("Post-pass statement
executed")
else:
    pass

print(x)

# Output:
# Pre-pass statement executed
# Post-pass statement executed
# 20
```

23. range of integers

The `range()` function generates a sequence of immutable numbers, commonly used to control the number of iterations in a **for** loop.

1. **Arguments:** It accepts up to three **integer** arguments: **range**([start], stop, [step]).
2. **Exclusivity:** The sequence generated is **inclusive of the start** value (default 0) but **exclusive of the stop** value. The loop runs up to, **butnotincluding**, the stop number.
3. **Data Type Restriction:** All arguments passed to **range**() **must be integers**. Using a **float** (e.g., **range**(**1.5**)) will result in a **TypeError**.
4. **Step:** The step argument (default 1) can be positive (counting up) or negative (counting down).

Example:

python

```
# One argument: start=0 (default),
stop=5, step=1 (default)
for i in range(5):
    print(i)

# Three arguments: start=2, stop=10
(exclusive), step=2
for j in range(2, 10, 2):
    print(j)

# Output:
# 0
# 1
# 2
# 3
# 4
# 2
# 4
# 6
# 8
```

22. Iteration: while loop, for loop

Iteration structures allow a block of code to be executed repeatedly.

A. `while` Loop

The `while` loop is a **condition-controlled** loop that executes a code block repeatedly as long as its specified condition remains **True**.

1. **Control:** The loop condition is checked at the start of every cycle.
2. **Responsibility:** The programmer **must manually** handle the update of the control variable to ensure the loop eventually terminates, preventing an **infinite loop**.
3. **Use Case:** Ideal when the number of repetitions is **unknown** beforehand.

Example (while loop):

python

```
count = 0
while count < 3:
    print(count)
    count = count + 1

# Output:
# 0
# 1
# 2
```

B. `for` Loop and Iteration Protocol

The `for` loop is an **iterator-controlled** loop, designed to traverse the items of any iterable object (list, range, string, etc.) until no elements are left.

a. **Iterable:** An object (like a list or range) that can be iterated over. It defines the \mathbf{__iter__} method.
b. **iter() Function:** This built-in function is called on an **iterable** (the container) and returns an **iterator** object. The iterator remembers the current position.
c. **next() Function:** This built-in function is called on the **iterator** to retrieve the next available item in the sequence.
d. **Termination:** When `next()` is called but there are no more items, it raises a **StopIteration** exception. The **for** loop handles this error internally to terminate gracefully.

Example (for loop & Iteration Protocol):

python

```
data = ["A", "B"]

# 'for' loop uses the protocol
automatically
for item in data:
    print(item)

# Manual simulation of the protocol
it = iter(data)
print(next(it))
print(next(it))

# Output:
# A
# B
# A
# B
```

25. Assignment Operators

Assignment operators are used to assign a value to a variable. The simplest is the basic assignment operator (=), but compound assignment operators perform an operation and an assignment simultaneously.

1. **Simple Assignment (=):** Assigns the value on the right to the variable on the left.
2. **Compound Assignment:** Shortcuts that combine an arithmetic or bitwise operation with assignment (e.g., + =, − =, * =, ** =, //=).
3. **Mechanism:** The statement $\mathbf{a+=5}$ is syntactic sugar for $\mathbf{a=a+5}$. It's often more efficient and always more concise.
4. **Immutability Detail:** For immutable types (like `int`), compound assignment often creates a **new** object. For mutable types (like `list`), operations like += might perform an **in-place** modification.

Example:

python

```
x = 10
y = 5

x += 3   # Same as: x = x + 3
y *= x   # Same as: y = y * x

print(x)
print(y)

# Output:
# 13
# 65
```

25. Operators: Identity, Ternary, and Membership

A. Identity Operators (`is`, `is not`)

Identity operators check if two variables refer to the **exact same object in memory**, not just if they have the same value.

1. **is:** Returns **True** if both variables point to the same memory address (same object), and **False** otherwise.
2. **is not:** Returns **True** if the variables refer to different objects.
3. **Difference from ==:** == checks for **value equality**; **is** checks for **object identity**. For small integers and short strings, Python often reuses the same memory object, so both might return True.

Example (Identity Operators):

python

```
a = [1, 2]
b = [1, 2]
c = a

print(a == b)
print(a is b)
print(a is c)

# Output:
# True
# False
# True
```

B. Ternary Conditional Operator

The Ternary Operator (or Conditional Expression) is a concise, single-line way to assign a value to a variable based on a condition.

1. **Syntax:** The structure is **[Value_If_True] if [Condition] else [Value_If_False]**.
2. **Function:** It's an **expression** that evaluates to a value, making it highly useful for assignments and function arguments. It is **not** a flow control statement like the `if/else` block.
3. **Short-Circuiting:** Like `if/else`, only one of the outcome expressions is evaluated.

Example (Ternary Operator):

python

```
age = 25
status = "Adult" if age >= 18 else
"Minor"
price = 10 if status == "Minor" else
20

print(status)
print(price)

# Output:
# Adult
# 20
```

C. Membership Operators (`in`, `not in`)

Membership operators test for the **presence** of a value within a collection (sequence).

1. **in:** Returns **True** if the element is found within the iterable (list, string, tuple, set, or dictionary key).
2. **not in:** Returns **True** if the element is **not** found within the iterable.
3. **Efficiency:** For Sets and Dictionaries, the membership check is highly efficient (average O(1) time complexity).

Example (Membership Operators):

python

```
data = [10, 20, 30]
word = "hello"

print(20 in data)
print(40 not in data)
print("l" in word)

# Output:
# True
# True
# True
```

26. break vs. continue

Both **break** and **continue** are loop control statements used to alter the normal execution flow of **for** and **while** loops.

1. **break:** Terminates the **entire loop** immediately. Execution jumps to the first statement **after** the loop body.
    - **Effect on else:** If the loop has an **else** block, the `break` statement **prevents** the **else** block from executing.
2. **continue:** Terminates **only the current iteration** of the loop. Execution jumps immediately to the top of the loop to begin the next iteration (checking the condition or fetching the next item).
3. **Use Cases:** `break` is used for **early exit** (e.g., finding a matching item). `continue` is used for **skipping** processing for specific conditions (e.g., skipping invalid data).

Example:

python

```
for i in range(5):
    if i == 2:
        continue
    if i == 4:
        break
    print(i)
else:
    print("Loop finished normally")

# Output:
# 0
# 1
# 3
```

27. Function

A function is a reusable block of code designed to perform a specific task.

1. **Definition:** Created using the **def** keyword, followed by the function name, parentheses **()**, and a colon **:**.
2. **Execution:** Functions are executed only when **called** by name.
3. **Namespace (Implicit):** Every function execution creates its own local **namespace** (scope) to hold its variables, ensuring they don't interfere with variables outside the function.
4. **Implicit Return (Crucial):** Every Python function **must** return a value. If no explicit **return** statement is provided, the function implicitly returns the special constant value **None**.

Example:

python

```
def greet(name):
    print("Hello,", name)
    # Implicitly returns None

result = greet("Alice")
print(result)

# Output:
# Hello, Alice
# None
```

28. Parameter

A parameter is a **name** listed in the function definition that acts as a placeholder for the values the function expects to receive during a call.

1. **Definition:** Parameters are defined within the parentheses `()` of the **`def`** statement.
2. **Scope:** Parameters are local variables within the function's scope.
3. **Types:** Can be defined as required (positional) or optional (default, variable-length, keyword-only).

29. Argument

An argument is the **actual value** passed to the function when it is called. Arguments correspond to the parameters defined in the function signature.

1. **Positional Arguments:** Matched to parameters based on their **order** (position).
2. **Keyword Arguments:** Matched to parameters based on their **name** (e.g., `func(name="Bob")`), allowing arguments to be passed out of order.
3. **Types (Brief):** Python supports various argument types:
    - **Default:** Arguments that have a fallback value if not provided.
    - **Variable-length (e.g., `*args`):** Collects extra positional arguments into a tuple.
    - **Keyword Arguments (e.g., `**kwargs`):** Collects extra keyword arguments into a dictionary.

The **return** statement sends a value back from the function to the place where the function was called.

1. **Single Return:** A function can explicitly return a single value of any type (e.g., **return x**).
2. **Implicit Return (Null Return):** If the **return** statement is omitted or **return** is used without an argument, the function returns the value **None** (This covers the "null return" concept).
3. **Multiple Return Values:** Python does **not** have true multiple return values. Instead, it packages multiple values (separated by commas) into a single **tuple** and returns that tuple. The caller can then use tuple **unpacking** to access the individual values.

Example (Return Values):

python

```
def calc_stats(a, b):
    # Multiple values returned as a
single tuple
    return a + b, a * b

s, p = calc_stats(10, 5)

print(s)
print(p)

# Output:
# 15
# 50
```

31. global

The **global** keyword is used **inside a function** to indicate that a variable being assigned to or modified is the **global variable** defined outside the function, rather than creating a new local variable.

1. **Scope and Namespace:** Python has a **Local scope** (inside the function) and a **Global scope** (outside all functions). Functions create their own **Local Namespace**.
2. **Access:** A function can always **read** a global variable without using the **global** keyword (read access is automatic).
3. **Modification:** To **modify** or **reassign** a global variable from within a function, the **global** keyword is **mandatory**. Without it, Python treats the assignment as creating a new variable in the local scope, leaving the global variable unchanged.

Example:

python

```
x = 10

def modify_global():
    # Declares intent to use the
global 'x'
    global x
    x = 50

def read_local():
    # Creates a NEW local variable
'x'
    x = 5
    print(x)

print(x)
modify_global()
print(x)
read_local()
print(x)

# Output:
# 10
# 50
# 5
# 50
```

32. Recursion

Recursion is a programming technique where a function calls **itself** one or more times to solve a problem. It's often used to solve problems that can be broken down into smaller, identical subproblems.

1. **Base Case (Mandatory):** Every recursive function **must** have a **base case**—a condition that stops the recursion. Without it, the function calls itself indefinitely, leading to a **RecursionError** (stack overflow).
2. **Recursive Step:** The part of the function where it calls itself, typically moving the problem closer to the base case.
3. **Efficiency Trade-off:** Recursion often results in clean, readable code (especially for mathematical problems like factorials or Fibonacci sequences) but can sometimes be **less memory-efficient** and **slower** than iteration (loops) due to the overhead of multiple function calls (stack frames).

Example: Factorial Calculation

python

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4))

# Output:
# 24
```

33. Lambda

A **lambda** function (also called an **anonymous function**) is a small, single-expression, inline function that doesn't require a standard function definition (`def`).

1. **Syntax:** Always defined using the **lambda** keyword: **lambda arguments: expression**.
2. **Restriction:** It can have any number of arguments but **must** contain only **one single expression**. The result of this expression is implicitly returned.
3. **Use Cases:** Primarily used where a small function is required for a short time, such as arguments to higher-order functions like **map()**, **filter()**, or **sorted()**.
4. **No Statements:** A lambda function cannot contain complex statements like `if`, `for`, `while`, or explicit `return`.

Example:

python

```
square = lambda x: x * x
print(square(5))

data = [1, 5, 8, 12]
odd_numbers = list(filter(lambda n: n
% 2 != 0, data))
print(odd_numbers)

# Output:
# 25
# [1, 5]
```

34. Closure

A closure is a **nested function** (a function defined inside another function) that **remembers and has access to the variables** from the outer function's local scope, **even after the outer function has finished execution**.

1. **Requirement:** A closure is created when an inner function **references** non-local variables from the outer function's scope.
2. **State Preservation:** The inner function "closes over" the outer function's state, preserving the values of those outer variables.
3. **Return:** The outer function must **return the inner function object**.
4. **Use Case:** Ideal for creating function factories and maintaining state between calls.

Example:

python

```
def make_multiplier(factor):
    def multiplier(n):
        return n * factor
    return multiplier

doubler = make_multiplier(2)
print(doubler(5))

tripler = make_multiplier(3)
print(tripler(5))

# Output:
# 10
# 15
```

35. Modules

A module is simply a Python file (`.py`) containing functions, classes, and variables.

1. **Purpose:** Modules are the fundamental way to organize and reuse code across different Python programs, preventing the need to rewrite the same logic.
2. **Built-in:** Python comes with a large Standard Library of built-in modules (e.g., `math`, `os`, `sys`).
3. **Namespace:** When a module is imported, its contents are placed into a separate **namespace**, which prevents naming conflicts with identifiers in your main program.

36. from

The `from` keyword is used to import **specific attributes** (functions, classes, or variables) directly from a module into the current namespace.

1. **Syntax:** `from [module] import [attribute]`
2. **Access:** Once imported this way, you can use the attributes directly, **without** prefixing them with the module name (e.g., `cos(x)` instead of `math.cos(x)`).
3. **Caution:** Using `from [module] import *` (importing all attributes) is generally discouraged as it can lead to namespace clashes if your file already defines names that exist in the module.

37. import

The **import** keyword is the primary way to bring all or part of a module into the current program's namespace.

1. **Syntax:** **import** [**module**]
2. **Access:** After importing, you must access the module's contents by prefixing them with the module's name (e.g., **math.sqrt()**). This keeps the namespace clean.
3. **Execution:** When a module is imported for the first time in a program's lifetime, its code is executed from top to bottom.

Example (Illustrating `from` and `import`):

python

```
import math
from random import randint

# Uses 'import': requires prefix
print(math.pi)

# Uses 'from': requires no prefix
print(randint(1, 10))

# Output:
# 3.141592653589793
# 7 (Example output)
```

38. as

The **as** keyword is used during an import to create an **alias** (a shorter or different name) for a module or a specific attribute.

1. **Purpose:** It improves code readability, saves typing (especially for long module names), and resolves name conflicts.
2. **Syntax:** **import [long_name] as [alias]** OR **from [module] import [attribute] as [alias]**
3. **Common Use:** Often used in data science (e.g., **import numpy as np**, **import pandas as pd**).

Example (as):

python

```
import collections as cl

queue = cl.deque([1, 2, 3])
queue.append(4)
print(queue)

# Output:
# deque([1, 2, 3, 4])
```

39. __name__

__name__ is a special built-in variable that automatically holds the **name of the current module** (i.e., the Python file currently being executed or imported).

1. **Value When Run Directly:** If a Python file is executed as the main program from the command line, its __name__ variable is set to the string literal __main__.
2. **Value When Imported:** If a Python file is imported as a module into another file, its __name__ variable is set to the module's actual file name (e.g., `'my_module'`).
3. **Use Case:** This allows code blocks to be selectively executed **only** when the file is run directly (as the main script) and prevents them from running when the file is simply being imported as a utility.

Example (__name__):

python

```
def main():
    print("Code runs when script is
executed directly")

if __name__ == "__main__":
    main()

# Output (if file is run via python
filename.py):
# Code runs when script is executed
directly
```

39. __name__

__name__ is a **special variable** (or **dunder name**) that holds the name of the current module. It is used to determine how the current file is being executed.

1. **Value When Run Directly:** If a Python file is executed as the main script from the command line, its **__name__** variable is set to the string literal **\mathbf{'__main__'}**.
2. **Value When Imported:** If the same file is imported as a module into another script, its **__name__** variable is set to the module's actual file name (e.g., **'my_module'**).
3. **Use Case:** The common **if __name__ ==' __main__'** block ensures that setup or execution code only runs when the file is the primary script, and is skipped when it's being used as a library.

Example:

python

```
def main():
    print("Running as main script")

if __name__ == "__main__":
    main()

# Output (if file is run via python
filename.py):
# Running as main script
```

44. Common Dunder Names (Special Variables)

Dunder names (short for **D**ouble **Under**score) are special identifiers used by Python for internal processing, attributes, and methods.

These are critical for understanding the mechanics of Python objects and modules.

Dunder Name	Purpose (Very Brief)
__name__	Holds the name of the module/script (\mathbf{'__main__' \text{ or module name}}).
__init__	Constructor method for class instantiation.
__str__	Defines the string representation of an object (for **print**()).
__repr__	Defines the unambiguous string representation of an object (for debugging).
__doc__	Holds the documentation string (docstring) for a module, class, or function.
__file__	The pathname of the file from which the module was loaded.

40. Top-Level Module

A **top-level module** refers to the module that is loaded or executed first when a Python program starts.

1. **Definition:** When you run a Python file directly from the command line (e.g., `python script.py`), that file acts as the top-level module for the entire execution.
2. **Special Status:** The top-level module is unique because the Python interpreter automatically sets its special variable, **__name__**, to the string value **\mathbf{'__main__'}**.
3. **Code Entry Point:** This special status is the reason for the common Python idiom: **if __name__ ==' __main__'**. This block ensures that certain code (like the primary logic or function calls) only executes when the script is run as the entry point, and not when it's simply imported as a library by another module.
4. **Contrast:** Any module that is loaded via an **import** statement is **not** the top-level module; its **__name__** variable will hold its actual file name.

Example:

python

```
# Assume this file is named 'app.py'

def run_application():
    print("Application Logic
Running")

if __name__ == "__main__":
    run_application()

# Output (if run via 'python
app.py'):
# Application Logic Running
```

41. Avoiding Circular Imports

Circular import (or cyclic dependency) is a common issue in Python that occurs when **Module A imports Module B, and Module B simultaneously imports Module A**. This creates a closed loop dependency.

1. **The Problem:** When the Python interpreter tries to load Module A, it starts executing its code. When it hits the `import B` statement, it pauses A and tries to load B. When it hits the `import A` statement inside B, it finds that A is not yet fully loaded, leading to an incomplete namespace. This usually results in a **NameError** because B tries to use an attribute from A that hasn't been defined yet, or vice versa.
2. **Symptoms:** You might see errors like **AttributeError** or **NameError** during the import stage.
3. **Solutions (How to Avoid Them):**
   - **Refactor the Code:** This is the best solution. Move the shared, necessary logic or constants into a **third, independent module (Module C)**. Both A and B can safely import C without creating a cycle.
   - **Import Locally:** If a module or function is only needed within a specific function (and not at the top level), move the **import** statement *inside* that function. This delays the loading until the function is actually called, often breaking the cycle.
   - **Conditional Import (Rare):** Use the **from [module] import [name]** syntax selectively instead of importing the entire module.

Example (Local Import Solution):

python

```
# Assume file 'a.py'
import b

def func_a():
    # Calling a function from module
b
    b.func_b()
    print("A executed")

# Assume file 'b.py' (Circular import
avoided by moving 'a' import)
def func_b():
    # If the import 'import a' was
here, it would be fine IF
    # func_b was guaranteed not to be
called during the initial import of
'b'.
    print("B executed")
```

42. Common Python Errors and Fixes

Errors fall into two main categories: **Syntax Errors** (language structure is violated) and **Exceptions** (errors detected during execution). Below are common exceptions related to type and data.

1. **TypeError**

Occurs when an operation or function is applied to an object of an **inappropriate data type**.

- **Cause Examples:** Trying to concatenate a string and an integer (e.g., `'Age is ' + 20`).
- **Fixes:** Explicitly convert the incompatible type using **str()** or **int()**. Ensure the function is called with the correct number and type of arguments.

2. **ValueError**

Occurs when an operation receives an argument of the **correct type** but an **inappropriate value**.

- **Cause Examples:** Trying to convert a non-numeric string to an integer (e.g., **int('hello')**). Trying to unpack an iterable into the wrong number of variables.
- **Fixes:** Use **try – except** blocks for validation, or perform explicit checks before conversion. Ensure the number of variables matches the number of elements being unpacked.

3. **NameError**

Occurs when a local or global variable name is **not found** in the current scope.

- **Cause Examples:** Misspelling a variable or function name, or trying to access a variable before it has been assigned a value.
- **Fixes:** Correct the spelling. Check the variable's scope (global vs. local) and ensure assignment occurs before use.

4. **IndexError** and **KeyError**

- **IndexError:** Occurs when an index is **out of range** for a sequence (list, tuple, string).
  - **Fix:** Use **len()** to check the boundaries or use defensive slicing.
- **KeyError:** Occurs when you try to access a dictionary element using a **non-existent key**.
  - **Fix:** Use the **in** operator to check for key existence, or use the safer **dict.get('key', default)** method.

Example (TypeError Fix):

python

```
number = 5
text = "Result: "

print(text + str(number))

# Output:
# Result: 5
```

43. Unexpected Python Types

This explores common Python objects that yield unexpected or interesting classes when inspected with **type()**.

- **Functions are Objects:** When you set a variable **x** equal to a function name **without parentheses** (e.g., $x = my\_function$), **x** holds the **function object** itself.
  - Type Result: $<class\ 'function'>$
- **The None Value:** The special constant **None** is an instance of its own unique class.
  - Type Result: $<class\ 'NoneType'>$
- **Lambda Functions:** Although anonymous, their type is the same as a standard defined function.
  - Type Result: $<class\ 'function'>$
- **The range Object:** **range()** is a memory-efficient sequence generator, not a list.
  - Type Result: $<class\ 'range'>$
- **Iterators:** When you start a **for** loop or use **iter()** on a list, Python creates a separate, position-tracking object.
  - Type Result: $<class\ 'list\_iterator'>$ (or similar, depending on the sequence)

Example: Inspecting a Function

python

```
def my_custom_func():
    return 1

x = my_custom_func
y = my_custom_func()

print(type(x))
print(type(y))

# Output:
# <class 'function'>
# <class 'int'>
```

More shorter

That syllabus is **chaotic**! 🤪

It's funny because it teaches **advanced concepts** (like **Closure**) **before beginner basics** (like **Lists**). It's non-sequential and jumbled, mixing expert topics (like **Dunder Names**) with the simplest fundamentals. It makes no logical sense!