Different types of Variables

There are 4 types of variables based on the context of their **scope** (where they can be accessed) and **lifetime** (how long they exist) within a program's execution

In Python, the primary types of variables from a *declaration and scope* perspective are:

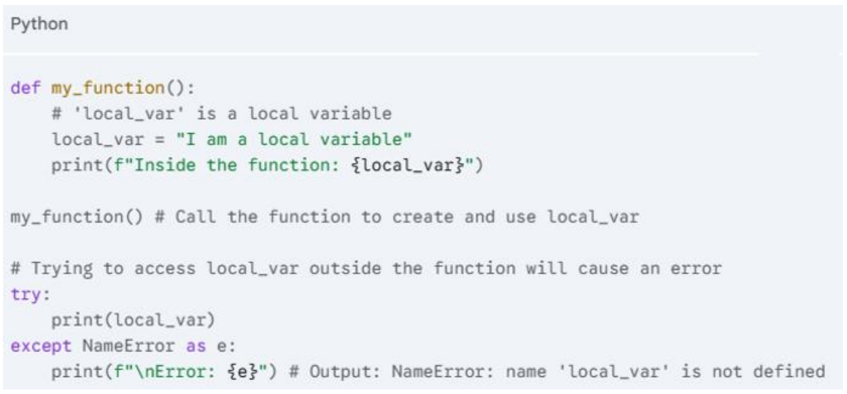
- **Local Variables**
- **Global Variables**
- **Nonlocal Variables** (less common for beginners, but important in specific scenarios)
- **Built-in Variables**

Let's break them down:

1. Local Variables

- **Definition:** Variables defined *inside* a function.
- **Scope:** They are accessible *only within the function* where they are defined. They cannot be accessed from outside that function.
- **Lifetime:** They are created when the function is called and destroyed (memory released) when the function finishes execution (returns or raises an exception).
- **Purpose:** Used for temporary data storage needed only for the duration of the function's execution.

Example:



Python

```python
def my_function():
    # 'local_var' is a local variable
    local_var = "I am a local variable"
    print(f"Inside the function: {local_var}")

my_function() # Call the function to create and use local_var

# Trying to access local_var outside the function will cause an error
try:
    print(local_var)
except NameError as e:
    print(f"\nError: {e}") # Output: NameError: name 'local_var' is not defined
```

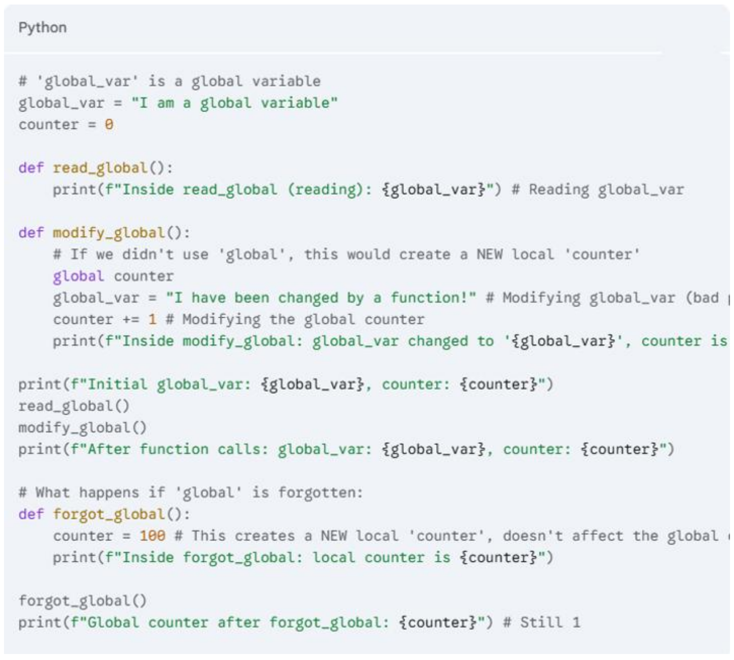
2. Global Variables

- **Definition:** Variables defined *outside* any function, typically at the top level of a Python script or module.
- **Scope:** They are accessible throughout the entire program, both inside and outside functions.
- **Lifetime:** They are created when the program starts and destroyed when the program finishes execution.
- **Purpose:** Used for data that needs to be shared across multiple functions or maintained throughout the program's lifetime (e.g., configuration settings, counters).

How to Use/Modify:

- You can *read* a global variable from inside a function without any special declaration.
- However, if you want to *modify* a global variable from inside a function, you **must** use the global keyword. If you don't, Python will assume you are creating a new *local* variable with the same name.

Example:



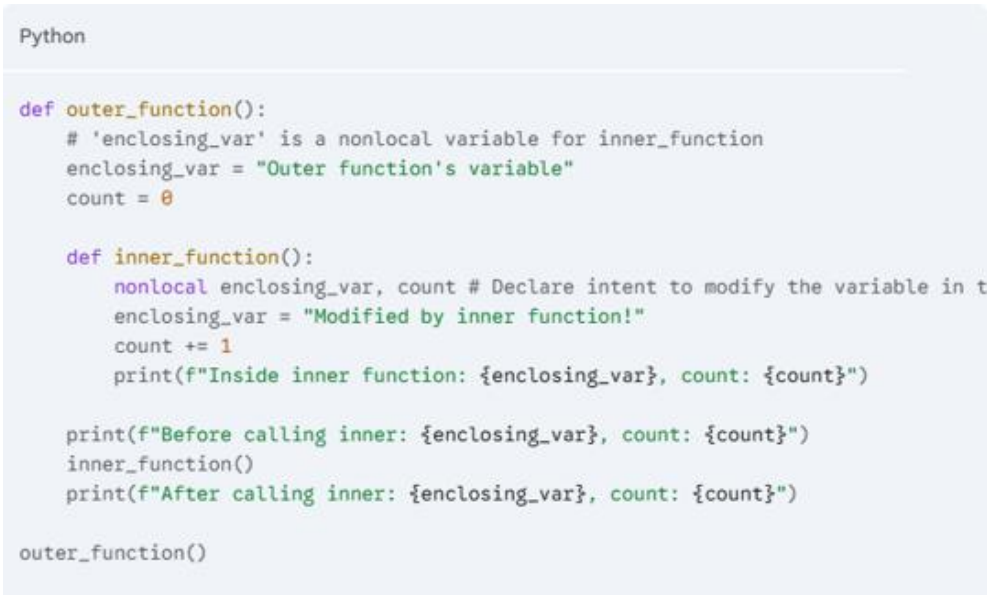
Python

```python
# 'global_var' is a global variable
global_var = "I am a global variable"
counter = 0

def read_global():
    print(f"Inside read_global (reading): {global_var}") # Reading global_var

def modify_global():
    # If we didn't use 'global', this would create a NEW local 'counter'
    global counter
    global_var = "I have been changed by a function!" # Modifying global_var (bad
    counter += 1 # Modifying the global counter
    print(f"Inside modify_global: global_var changed to '{global_var}', counter is

print(f"Initial global_var: {global_var}, counter: {counter}")
read_global()
modify_global()
print(f"After function calls: global_var: {global_var}, counter: {counter}")

# What happens if 'global' is forgotten:
def forgot_global():
    counter = 100 # This creates a NEW local 'counter', doesn't affect the global
    print(f"Inside forgot_global: local counter is {counter}")

forgot_global()
print(f"Global counter after forgot_global: {counter}") # Still 1
```

3. Nonlocal Variables

- **Definition:** Variables defined in an *enclosing (outer) function*, but not in the global scope. They exist when you have nested functions.
- **Scope:** Accessible within the nested (inner) function(s).
- **Lifetime:** They persist as long as the enclosing function's scope is active.
- **Purpose:** Used to allow a nested function to modify a variable in its immediately enclosing scope (not the global scope).

Example:



Python

```python
def outer_function():
    # 'enclosing_var' is a nonlocal variable for inner_function
    enclosing_var = "Outer function's variable"
    count = 0

    def inner_function():
        nonlocal enclosing_var, count # Declare intent to modify the variable in t
        enclosing_var = "Modified by inner function!"
        count += 1
        print(f"Inside inner function: {enclosing_var}, count: {count}")

    print(f"Before calling inner: {enclosing_var}, count: {count}")
    inner_function()
    print(f"After calling inner: {enclosing_var}, count: {count}")

outer_function()
```

4. Built-in Variables

- **Definition:** Pre-defined variables (and functions) that are always available when you run Python. They reside in the builtins module.
- **Scope:** Global to all modules and functions by default.
- **Lifetime:** Exist throughout the entire Python session.
- **Examples:** True, False, None, __name__ (often used to check if a script is being run directly), __file__, __doc__.

Example:

```Python
print(f"This script's name: {__name__}")
if __name__ == "__main__":
    print("This code runs when the script is executed directly.")

print(f"The value of True is: {True}")
print(f"The value of None is: {None}")
```

Understanding these different variable types based on their scope is crucial for writing correct, readable, and maintainable Python code, especially as your programs grow in complexity and use multiple functions.