**Python Basics Worksheet for Beginners (Week1)**

**1. Python Coding Style and Naming Conventions**

Python has a set of widely accepted **coding style guidelines** that make code more readable and maintainable. The most famous style guide is **PEP 8**, which suggests best practices for formatting and naming in Python. Here are some key conventions:

* **Indentation:** Use 4 spaces (not tabs) for each level of indentation in code blocks (such as inside loops, functions, and classes).
* **Line length:** Try to keep lines under 79 characters for readability (for beginners, this means avoid writing very long single-line statements).
* **Naming variables and functions:** Use lowercase letters with words separated by underscores (this is known as *snake\_case*). For example, user\_name or total\_amount.
* **Naming classes:** Use CapitalizedWords (also known as *PascalCase* or *CamelCase*). For example, BankAccount or StudentRecord.
* **Naming constants:** Use all uppercase letters with underscores for constants (e.g., MAX\_RETRIES = 5).
* **Spacing:** Put a single space around operators and after commas for readability, e.g. x = 10, y = x + 5.
* **Comments:** Use comments (# ...) to explain non-obvious parts of code. Keep comments up-to-date and concise.

Following these conventions makes your code easier to read for yourself and others. Remember that consistency is key — pick a style and stick to it throughout your code.

**Example (bad vs. good style):**

# Bad Style Example

x=5

if(x>0):

print("Positive number")

# Good Style Example

number = 5

if number > 0:

print("Positive number")

In the bad style example, there are issues like no spaces around operators, unnecessary parentheses in the if, and a non-descriptive variable name. The good style example fixes these issues according to the conventions listed above.

**Exercise 1:** The following code has poor style and naming. Rewrite it using proper Python style and naming conventions:

# messy code

Amt= 100

DEF calc\_tax(amount):

rate=0.1

Tax= amount\*rate

print("tax:",Tax)

calc\_tax(Amt)

*Hint:* Fix the indentation, rename variables and functions to follow snake\_case for regular names and PascalCase for class names (though there are no classes here), and add spaces around operators and after commas as needed.

**2. Variables and Basic Data Types**

A **variable** is a name that refers to a value. In Python, you create a variable by assigning a value to it using the = sign. Python is **dynamically typed**, which means you don't need to declare the type of a variable; the type is determined by the value it holds. Here are some basic data types in Python:

* **Integer (int):** Used for whole numbers (e.g., 5, -20).
* **Floating-point (float):** Used for decimal or fractional numbers (e.g., 3.14, 0.5).
* **String (str):** Used for text, enclosed in quotes (e.g., "Hello, world!" or 'Python').
* **Boolean (bool):** Logical values True or False (note the capital T and F).
* **List (list):** An ordered, mutable collection of items (e.g., [1, 2, 3] or ["apple", "banana"]).
* **Tuple (tuple):** An ordered, immutable collection of items (e.g., (10, 20, 30)).
* **Dictionary (dict):** An unordered collection of key-value pairs (e.g., {"name": "Alice", "age": 30}).
* **NoneType (None):** Represents the absence of a value (None is a special keyword for "no value").

You can use the type() function to check the type of a variable. For example, type(42) returns int and type("42") returns str.

**Example: Creating and using different types of variables**

# Numbers

age = 30 # int

price = 19.99 # float

# String

name = "Alice" # str

# Boolean

is\_member = True # bool

# List

fruits = ["apple", "banana", "cherry"] # list of strings

fruits.append("durian") # add an item to the list

print(fruits) # Output: ['apple', 'banana', 'cherry', 'durian']

# Tuple

dimensions = (1920, 1080) # tuple of two ints (width, height)

print(dimensions[0]) # Output: 1920

# Dictionary

person = {"name": "Bob", "age": 25}

person["city"] = "New York" # add a new key-value pair to the dictionary

print(person) # Output: {'name': 'Bob', 'age': 25, 'city': 'New York'}

# NoneType

result = None

print(result) # Output: None

In this example, we created variables of various types and performed some operations:

* Added an item to a list using append.
* Accessed an element of a tuple by index (tuples are like lists but cannot be changed after creation).
* Added a new key-value pair to a dictionary.
* Used None to indicate an "empty" or uninitialized value.

**Real-world scenario:** A dictionary is often used to represent structured data (like a simple database record or JSON data). For example, person above could represent a record from a users database. Lists could represent collections of items like a shopping list.

**Exercise 2:** Create variables of the following types and assign appropriate values to them:

1. An integer variable named months with the value 12.
2. A float variable named temperature with the value 36.6.
3. A string variable named message with the value "Hello".
4. A list variable named colors containing three color names as strings.
5. A dictionary variable named book with keys "title", "author", and "year" representing a book's information.

After creating them, print each variable to confirm their values.

**3. Python Naming Rules**

Python has strict rules for what constitutes a valid name (also called an identifier) for variables, functions, classes, etc. It's important to know these **naming rules** to avoid syntax errors:

* Names can only contain **letters**, **digits**, and **underscores** (\_).
* Names **cannot start with a digit**. They must start with a letter or an underscore.
* Names are **case-sensitive**. For example, myVar and myvar would be two different variables.
* You **cannot use Python's reserved keywords** as names. Keywords are special words that are part of the Python syntax (like if, for, while, class, True, False, etc.). For example, you cannot name a variable if or def. (Hint: To see all keywords, you can use import keyword; print(keyword.kwlist) in Python.)
* Avoid using built-in function names for your variables (like print, list, str) because it will override the built-in, causing potential confusion or errors.
* Names can start with an underscore \_. This is allowed and sometimes used to indicate a name is meant for internal or private use. For example, \_secret\_value is a valid name. But as a beginner, you typically won't need to start names with underscore.

Some examples of **valid names**: x, my\_variable, name2, \_temp, MAX\_VALUE  
Some examples of **invalid names** (and why they are invalid):

* 2nd\_player (starts with a digit)
* user-name (contains a hyphen -, which is not allowed)
* class (this is a reserved keyword in Python)
* my var (contains a space, not allowed in names)
* break (reserved keyword)

**Exercise 3:** Which of the following are valid Python variable names? For those that are not valid, explain why.

* total\_cost
* 3d\_model
* playerOne
* def
* email\_address
* first-name
* PI

*(Write your answers in words. This is not a coding exercise but a check of understanding.)*

**4. Data Type Conversions**

Often you will need to convert data from one type to another. Python provides built-in functions for **type conversion** (also called type casting). Common conversions include:

* int(x) – Convert x to an integer (e.g., int("42") becomes 42). If x is a floating-point number, it will truncate the decimal part (e.g., int(3.9) becomes 3).
* float(x) – Convert x to a float (e.g., float("2.5") becomes 2.5, and float(5) becomes 5.0).
* str(x) – Convert x to a string (e.g., str(123) becomes "123").
* bool(x) – Convert x to a boolean. Most values convert to True except: 0, 0.0, '' (empty string), None, and empty collections (which all become False).
* list(x) – Convert x to a list. For example, list("hello") becomes ['h', 'e', 'l', 'l', 'o'], and list((1,2,3)) becomes [1, 2, 3].
* tuple(x) – Convert x to a tuple (e.g., tuple([1, 2, 3]) becomes (1, 2, 3)).
* dict(x) – There is a dict() constructor which can convert certain sequences of pairs into a dictionary, but that's more advanced. Typically, you'll create dicts directly rather than converting other types to dict.

**Why convert types?** Suppose you accept input from a user via the input() function. input() returns a string, even if the user typed a number. If you want to perform numerical calculations, you'll have to convert that string to an int or float. Similarly, if you want to concatenate a number with a string for display, you need to convert the number to a string.

**Example: Converting types**

# Start with a string that contains digits

num\_str = "50"

# Convert the string to an integer to do arithmetic

num\_int = int(num\_str)

result = num\_int + 10

print(result) # Output: 60 (integer arithmetic)

# Convert a number to a string to concatenate with another string

days = 7

message = "I have " + str(days) + " days off work."

print(message) # Output: I have 7 days off work.

# Convert a float to int (this will lose the fractional part)

price = 19.99

whole\_price = int(price)

print(whole\_price) # Output: 19

# Boolean conversions

print(bool(0)) # Output: False

print(bool(42)) # Output: True

print(bool("")) # Output: False (empty string is False)

print(bool("hi")) # Output: True (non-empty string is True)

Notice in the examples:

* We converted "50" (string) to 50 (int) to add 10.
* We converted the integer 7 to "7" (string) to concatenate it into a sentence.
* Converting 19.99 to int gave 19 (decimal part dropped).
* Boolean conversion treats "empty" values as False.

**Exercise 4:** The following variables are given:

length\_str = "15"

width\_str = "8.4"

They represent the length and width of a rectangle as strings. Convert these to appropriate numeric types (int or float) and calculate the area of the rectangle. Store the area in a variable area and print it.

*Hint:* You might need to use both int() and float() depending on the data.

**5. Membership Operator (in)**

The **membership operator** in is used to check if a value is a member of a sequence (such as a string, list, tuple, or set) or a key in a dictionary. It returns a boolean True or False. There is also a negated version, not in, to check if something is *not* a member.

How in works for different types:

* **String:** Checks if a substring exists within another string.
* **List/Tuple:** Checks if an element is present in the list or tuple.
* **Dictionary:** Checks if a key exists in the dictionary (it does *not* check the values by default).

**Example: Using in**

# String membership

sentence = "The quick brown fox"

print("quick" in sentence) # Output: True (substring "quick" is found)

print("slow" in sentence) # Output: False

# List membership

numbers = [2, 4, 6, 8]

print(4 in numbers) # Output: True

print(5 in numbers) # Output: False

# Tuple membership

letters = ('a', 'b', 'c')

print('d' in letters) # Output: False

print('a' in letters) # Output: True

# Dictionary membership

student = {"name": "Eva", "grade": "A", "age": 20}

print("name" in student) # Output: True (checks keys)

print("Eva" in student) # Output: False (string "Eva" is not a key, it's a value)

print("age" not in student) # Output: False ("age" is a key, so "age" in student is True, hence "age" not in student is False)

In these examples:

* "quick" in sentence checks for substring in a string.
* 4 in numbers checks for element in list.
* 'a' in letters checks for element in tuple.
* "name" in student checks if "name" is a key in the dictionary student.

**Real-world usage:** You might use in to check if a user input exists in a list of valid options, or if a certain field exists in a dictionary representing JSON data.

**Exercise 5:** Given the following data, use the in operator to answer the questions:

allowed\_users = ["alice", "bob", "charlie"]

email = "bob@example.com"

info = {"email": "bob@example.com", "age": 25, "member": True}

1. Check if the string "bob" is in the allowed\_users list.
2. Check if the substring "@" is in the email string.
3. Check if the key "age" is in the info dictionary.
4. Check if the value False is in the info dictionary.

*(Print out True or False for each of the above checks.)*

**6. Index Notation, Range, and Slice Operator**

Python uses **index notation** to access individual elements of sequences like strings, lists, and tuples. Indexes are zero-based, meaning the first element is index 0, the second is index 1, and so on. You use square brackets [] to index:

my\_list = ['a', 'b', 'c', 'd']

first\_item = my\_list[0] # 'a'

second\_item = my\_list[1] # 'b'

Negative indexes count from the end:

last\_item = my\_list[-1] # 'd' (last element)

second\_last = my\_list[-2] # 'c'

Python also provides a powerful **slice operator** using the colon : inside square brackets to get a subsequence (slice) of a sequence. The syntax is sequence[start:stop], which gets elements from index start up to but not including index stop. You can also add an optional step: sequence[start:stop:step].

* If you omit start, slicing starts from the beginning.
* If you omit stop, slicing goes until the end.
* The step is how many indices to jump each time (default is 1).

**Examples: Slicing a list and string**

letters = ['a','b','c','d','e','f']

print(letters[1:4]) # Output: ['b', 'c', 'd'] (indexes 1,2,3)

print(letters[:3]) # Output: ['a', 'b', 'c'] (start at beginning, up to index 2)

print(letters[4:]) # Output: ['e', 'f'] (from index 4 to end)

print(letters[-3:]) # Output: ['d', 'e', 'f'] (last 3 elements)

print(letters[::2]) # Output: ['a', 'c', 'e'] (every 2nd element, step=2)

print(letters[::-1]) # Output: ['f', 'e', 'd', 'c', 'b', 'a'] (step=-1 reverses the list)

text = "Hello World"

print(text[0:5]) # Output: Hello (characters at indexes 0-4)

print(text[6:]) # Output: World (characters from index 6 to end)

In these examples:

* letters[1:4] gets a slice from index 1 to 3.
* letters[:3] gets the first three elements.
* letters[-3:] gets the last three elements.
* letters[::2] gets every second element (skips one each time).
* letters[::-1] returns a reversed copy of the list.
* For the string text, text[0:5] gives "Hello" and text[6:] gives "World".

**Range**: In Python, range() is a function that generates a sequence of numbers. It's often used with loops. For example, range(5) generates numbers 0,1,2,3,4. We will see range() in action in the looping section. For indexing and slicing, range() isn't directly used, but it's good to be aware that range(n) is another way to produce a sequence (in this case of integers 0 to n-1).

**Exercise 6:** Consider the list data = [10, 20, 30, 40, 50, 60] and the string s = "abcdefg". Write expressions or lines of code to achieve the following:

1. Get the first three elements of data using slicing.
2. Get the last two elements of data using slicing.
3. Get every other element of data (alternate elements) using slicing.
4. Get the substring "cde" from the string s using slicing.
5. Reverse the string s using slicing.

*(Print the results of each to verify your answers.)*

**7. Control Flow: if, elif, else**

**Control flow** statements allow your program to make decisions and execute certain blocks of code only if certain conditions are met. The primary way to do this is with the if statement in Python. The syntax is:

if condition:

# code block to execute if condition is True

elif another\_condition:

# code block if the first condition was False but this one is True

else:

# code block if all above conditions are False

Key points:

* The condition is an expression that evaluates to True or False (or a truthy/falsy value).
* **Indentation** is used to define the blocks under if/elif/else. Each block must be consistently indented (typically 4 spaces).
* elif (short for "else if") and else are optional. You can have just an if by itself, or an if with one or more elif parts and an optional final else.

Common operators used in conditions include comparison operators (==, !=, >, <, >=, <=) and logical operators (and, or, not). For example, if x > 0 and x < 10: checks if x is between 1 and 9 (inclusive).

**Example: Using if-elif-else**

age = 17

if age < 13:

print("You are a child.")

elif age < 18:

print("You are a teenager.")

else:

print("You are an adult.")

# Output: "You are a teenager."

In this example, the program checks the age and prints a message depending on which range the age falls into. Since age is 17, the first condition (age < 13) is False, the elif condition (age < 18) is True, so "You are a teenager." is printed, and the rest is skipped.

**Another Example: Simple if and if-else**

# Check if a number is positive

num = 5

if num > 0:

print("Positive")

# Check if a number is even or odd

if num % 2 == 0:

print("Even")

else:

print("Odd")

Output:

Positive

Odd

Explanation:

* The first if checks if num is greater than 0. Because 5 > 0 is True, it prints "Positive". There is no else for this if, so if the condition were False, nothing would happen in that part.
* The second if uses the modulus operator % to check if the remainder of num divided by 2 is 0 (which means the number is even). 5 % 2 is 1 (not 0), so the condition is False, and the else part executes, printing "Odd".

**Real-world scenario:** You might use if statements to check user input and provide feedback (e.g., if a password is correct or not), to ensure certain conditions are met before running some code (e.g., if a file exists), or to categorize data (like grading scores, handling different cases in business logic, etc.).

**Exercise 7:** Write an if-elif-else chain that determines the state of water based on a temperature value in the variable temp\_c (temperature in Celsius):

* If temp\_c is below 0, print "Frozen (ice)"
* Elif temp\_c is above 100, print "Gas (steam)"
* Else (otherwise between 0 and 100 inclusive), print "Liquid water"

Try testing your code with temp\_c = -5, temp\_c = 25, and temp\_c = 105 to see the different outputs.

**8. Looping with for and while**

Loops allow you to repeat a block of code multiple times. Python provides two main loop structures: **for loops** and **while loops**.

**For Loops**

A **for loop** is used to iterate over a sequence (such as a list, tuple, string, or range of numbers). The syntax is:

for variable in sequence:

# code block to execute for each element in sequence

On each iteration, the variable takes the value of the next element in the sequence, and the loop body executes with that value.

A common use of for is with the range() function to loop a certain number of times. For example:

for i in range(5):

print(i)

This will print 0, 1, 2, 3, 4 each on a new line (range(5) produces 0 through 4).

**Example: Iterating through a list**

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print("I like", fruit)

Output:

I like apple

I like banana

I like cherry

**While Loops**

A **while loop** repeats as long as a certain condition is True. The syntax is:

while condition:

# code block to execute repeatedly

# make sure something in this block will eventually make the condition False

If the condition never becomes False, a while loop can run forever, so be careful to update variables or use break to exit the loop at some point.

**Example: Using a while loop**

count = 5

while count > 0:

print("Counting down:", count)

count -= 1 # decrease count by 1 each time

print("Blast off!")

Output:

Counting down: 5

Counting down: 4

Counting down: 3

Counting down: 2

Counting down: 1

Blast off!

In this example, the loop runs while count > 0. We decrement count each time, so eventually count becomes 0 and the loop stops. Then "Blast off!" is printed after the loop.

**Loop Control: break and continue**

* break can be used inside a loop to exit the loop immediately. For example, you might search for something in a list and break once you find it.
* continue skips the rest of the loop body for the current iteration and moves to the next iteration.

**Example: Using break and continue**

# Find the first even number in a list

numbers = [1, 3, 5, 4, 7, 8]

for n in numbers:

if n % 2 == 0:

print("Found an even number:", n)

break # exit loop when first even number is found

# Output: Found an even number: 4

# Skip printing odd numbers

for n in numbers:

if n % 2 != 0:

continue # skip odd numbers

print("Even number:", n)

# Output:

# Even number: 4

# Even number: 8

In the break example, the loop stops at the first even number (4). In the continue example, the print runs only for even numbers (skipping any iteration where the number is odd).

**Exercises 8 and 9:** Practice with loops:

* **Exercise 8:** Using a **for loop**, calculate the sum of all numbers in the list values = [3, 7, 1, 12, 9]. Print the sum after the loop.
* **Exercise 9:** Using a **while loop**, print the numbers 1 through 5 (inclusive). (Hint: Initialize a counter variable to 1, and loop while it’s <= 5, printing the counter and incrementing it by 1 each iteration.)

**9. Functions and Parameters**

Functions allow you to reuse and organize code into logical blocks. A **function** is defined using the def keyword, giving it a name and optional parameters. After defining a function, you can "call" it (execute it) whenever needed.

Syntax for defining a function:

def function\_name(param1, param2=default\_value, \*args, \*\*kwargs):

"""

Optional docstring (description of function).

"""

# code block

return result # optional return statement

Key points:

* Use a descriptive function name (follow naming rules: lowercase\_with\_underscores).
* Parameters are variables that will receive the values (arguments) you pass into the function.
* You can provide default values for parameters (e.g., param2=default\_value above), making them optional.
* \*args and \*\*kwargs are advanced (for variable number of arguments); beginners can ignore these for now or know that they exist.
* The return statement is used to send a result back to where the function was called. If a function doesn't have a return statement, it returns None by default after executing.

**Forcing data types for parameters:** Python is dynamically typed and does not enforce data types of arguments at runtime. However, you can **hint** types using function annotations, and you can manually check types inside the function if necessary.

* *Type hints* (Python 3.5+): You can write a function like def greet(name: str) -> None: to indicate name should be a string and the function returns nothing. These are not enforced by Python at runtime, but they serve as documentation and can be checked by external tools (like linters or IDEs).
* *Manual type checking:* Inside your function, you could use isinstance() to check a parameter's type and perhaps raise an error if it's not what you expect. This is rarely needed for beginners but is one way to "force" a type.

**Example: Defining and calling functions**

# A simple function with one parameter and no return value

def greet\_user(name):

print(f"Hello, {name}!")

greet\_user("Alice") # Call the function with argument "Alice"

# Output: Hello, Alice!

# A function with a return value

def add(a, b):

return a + b

result = add(5, 7)

print(result) # Output: 12

# A function with a default parameter

def power(base, exponent=2):

"""Return base raised to the given exponent. Default exponent is 2."""

return base \*\* exponent

print(power(3)) # Output: 9 (3^2)

print(power(2, 3)) # Output: 8 (2^3)

# Function with type hints (not enforced, just for clarity)

def divide(x: float, y: float) -> float:

# Check types manually (optional)

if not isinstance(x, (int, float)) or not isinstance(y, (int, float)):

raise TypeError("x and y must be numbers")

return x / y

print(divide(10, 2)) # Output: 5.0

Let's break down the examples:

* greet\_user(name) prints a greeting. It uses an **f-string** (notice the f before the string and the {name} inside) for string formatting, which is a convenient way to include variable values in a string.
* add(a, b) returns the sum of a and b. After calling it with 5 and 7, we store the result and print it.
* power(base, exponent=2) has a default value for exponent. If we don't provide the exponent, it uses 2 by default (squaring the base). We call it once with only base 3, and once with 2 and 3 to cube 2.
* divide(x, y) demonstrates type hints and a manual type check. The type hints suggest x and y should be floats (or ints, since an int is also an instance of (int, float)), and that it returns a float. We included a check using isinstance to show how you could enforce that x and y are numbers, and raise a TypeError if not. In practice, one might not include such a check unless it's critical.

**Exercise 10:** Write a function called square() that takes one parameter (a number) and returns the square of that number. Include a type hint indicating the parameter should be a number (int or float) and the return type is a number. Then, call your function with a couple of examples (e.g., square(5), square(2.5)) and print the results.

*Bonus:* Inside the function, add a check to ensure the argument is an int or float, and print a friendly error message if not.

**10. Classes and Objects (Basic OOP)**

Python supports **Object-Oriented Programming (OOP)**, which allows you to define your own custom data types called **classes**. A class is like a blueprint for objects (instances of that class). You use classes to encapsulate data (attributes) and functions that operate on that data (methods).

Basic terminology:

* **Class:** A blueprint or template for objects. Defined with the class keyword.
* **Object (Instance):** A concrete occurrence of a class. You create an object by calling the class as if it were a function (e.g., obj = MyClass()).
* **Attributes:** Variables that belong to an object (or class). Attributes typically describe the object's state.
* **Methods:** Functions that belong to a class, defined inside the class, that typically act on the object's data.

**Defining a class:**

class Person:

def \_\_init\_\_(self, name, age):

# \_\_init\_\_ is the initializer (constructor) method that runs when a new object is created

self.name = name # assign the name attribute

self.age = age # assign the age attribute

def greet(self):

# A simple method that uses the object's attributes

print(f"Hello, my name is {self.name} and I am {self.age} years old.")

In the Person class above:

* \_\_init\_\_ is a special method that initializes new Person objects. The self parameter refers to the instance itself (it is automatically passed when you create a new object).
* We set self.name and self.age for the new object using the values provided.
* greet is a method that prints a greeting using the object's data.

**Using a class (creating objects):**

# Create instances of Person

p1 = Person("Alice", 30)

p2 = Person("Bob", 25)

# Access attributes

print(p1.name) # Output: Alice

print(p2.age) # Output: 25

# Call methods

p1.greet() # Output: Hello, my name is Alice and I am 30 years old.

p2.greet() # Output: Hello, my name is Bob and I am 25 years old.

**Another Example: A simple class with a method and attribute:**

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14159 \* (self.radius \*\* 2)

# Using the Circle class

c = Circle(5)

print("Radius:", c.radius) # Output: Radius: 5

print("Area:", c.area()) # Output: Area: 78.53975

In this example, Circle has an attribute radius and a method area() that calculates the area using the circle's radius. We create an instance c with radius 5 and call c.area().

**Real-world analogy:** Think of a class as a blueprint for a house. The blueprint (class) defines what properties (rooms, size) and capabilities (like functions of appliances) a house (object) will have. Each actual house built from that blueprint is an object; while they share the same design, each house has its own state (one house might have its rooms painted different colors than another, etc.).

**Exercise 11:** Define a class Book with two attributes: title and author (set these in the \_\_init\_\_ method). Add a method description that prints or returns a string like "<title>" by <author>. Create an instance of Book for your favorite book, and call the description method.

*(No need to implement complex logic — just practice creating a class, attributes, and a method.)*

**11. Shallow and Deep Copies**

When working with objects like lists or dictionaries, it's important to understand the difference between **shallow copy** and **deep copy** in Python.

* **Assignment (no copy):** If you just assign one variable to another, e.g. list\_b = list\_a, you are not making a new copy of the list. Both list\_a and list\_b will refer to the same list object in memory. Changes through one variable will affect the other.
* **Shallow copy:** Creates a new object, but does not recursively copy nested objects. In other words, it copies the outer container, but the elements inside are still references to the original elements. For a list, a shallow copy will create a new list, but if the list contains mutable objects (like other lists or dicts), it doesn't create copies of those inner objects.
* **Deep copy:** Creates a new object and **recursively** copies all objects inside it. That means it makes clones of nested objects all the way down, so the new structure is completely independent of the original.

In Python, you can create shallow copies in a few ways:

* For lists: using list\_copy = original\_list.copy() or list\_copy = original\_list[:] (slicing the whole list).
* For dictionaries: dict\_copy = original\_dict.copy() (shallow copy of a dict).
* Using the copy module:
* import copy
* copy.copy(obj) # returns a shallow copy of obj
* copy.deepcopy(obj) # returns a deep copy of obj

**Example: Assignment vs Shallow Copy vs Deep Copy**

import copy

# Original list with nested list

original = [1, 2, [3, 4]]

assigned = original # no copy, just another reference to the same list

shallow\_copy = original.copy() # shallow copy of the list

deep\_copy = copy.deepcopy(original) # deep copy of the list

# Modify the nested list in the original

original[2].append(5)

print("Original:", original) # Original: [1, 2, [3, 4, 5]]

print("Assigned:", assigned) # Assigned: [1, 2, [3, 4, 5]] (same object as original)

print("Shallow Copy:", shallow\_copy) # Shallow Copy: [1, 2, [3, 4, 5]]

print("Deep Copy:", deep\_copy) # Deep Copy: [1, 2, [3, 4]]

Explanation:

* We start with original = [1, 2, [3, 4]]. This is a list containing another list as its third element.
* assigned = original doesn't create a new list at all. It's just another name for the same list in memory.
* shallow\_copy = original.copy() creates a new top-level list, but its contents still reference the same inner list [3, 4] from the original.
* deep\_copy = copy.deepcopy(original) creates a completely independent copy, including a brand new inner list with values [3, 4].

After appending 5 to the inner list of original, we saw that:

* assigned reflects the change (because it's the same list as original).
* shallow\_copy also reflects the change in its inner list (because it shares the inner list with original).
* deep\_copy does not change (it has its own copy of the inner list).

**Why this matters:** If you want to copy complex objects without accidentally linking them, you may need a deep copy. But deep copies can be expensive in terms of performance. Often, knowing that a shallow copy still shares references is enough to avoid surprises (e.g., when copying a list of lists).

**Exercise 12:** Given the following code:

a = [100, 200, [300, 400]]

b = a # assignment, no real copy

c = a.copy() # shallow copy

1. What will happen to a, b, and c if you execute a[0] = 111?
2. What will happen to a, b, and c if you execute a[2].append(500)?

Write your answers explaining the state of a, b, and c after each operation. *(This is a thought exercise; you can also write a short script to verify.)*

**12. Anonymous Functions (Lambdas)**

Python supports creating **anonymous functions** using the lambda keyword. A lambda function is a small, one-line function that is defined without a name (hence "anonymous"). These are also sometimes called **lambda expressions**.

The syntax of a lambda:

lambda parameters: expression

The lambda returns the value of the expression. Lambdas are often used for short, simple functions that are passed as arguments to other functions or used in places where a full def function might be too verbose.

**Characteristics of lambda functions:**

* They can take any number of arguments (including optional ones) but have to contain **exactly one expression**.
* They implicitly return the value of that expression (no need for a return statement).
* They don't have a name, but you can assign them to a variable if you want to reuse them (though in that case, a regular def might be clearer).

**Where are lambdas used?** Commonly in sorting, filtering, or when working with functions like map, filter, and reduce. For example, if you want to sort a list of names by their length, you could use a lambda as the key function. Or if you want to apply a simple operation to every element of a list without defining a separate function.

**Examples:**

# Lambda for a simple arithmetic operation

double = lambda x: x \* 2

print(double(5)) # Output: 10

# Using a lambda in sorting (sort list of words by last letter)

words = ["apple", "banana", "cherry", "date"]

words.sort(key=lambda w: w[-1]) # sort by last character of each word

print(words) # Output: ['banana', 'apple', 'date', 'cherry']

# Using lambdas with filter to get even numbers from a list

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = list(filter(lambda n: n % 2 == 0, numbers))

print(even\_numbers) # Output: [2, 4, 6]

# Using lambdas with map to square each number in a list

squares = list(map(lambda n: n\*\*2, numbers))

print(squares) # Output: [1, 4, 9, 16, 25, 36]

In these examples:

* We defined double as a lambda that multiplies input x by 2. We then called it like a normal function.
* We used lambda w: w[-1] as a key to sort by the last character of each word in the list.
* We used filter with a lambda to keep only even numbers from the list.
* We used map with a lambda to square each number in the list.

Lambdas are handy for short functions, but if the operation is complex, it's better to define a normal function with def for clarity and reuse.

**Exercise 13:** Suppose you have a list of integers nums = [5, 8, 12, 15, 7, 1]. Write a line of code using filter and a lambda to create a new list greater\_than\_10 that contains only the numbers from nums that are greater than 10. Print the result.

*(Hint: The lambda should return True for numbers that should be kept.)*

**13. Namespaces and Scope (Visibility of Names)**

A **namespace** in Python is like a container that holds a set of names (variable names, function names, etc.) and their corresponding objects. Different namespaces are isolated, which helps avoid name collisions (two different things having the same name). **Scope** refers to the region of a program where a particular namespace is directly accessible.

Key points:

* When you create a variable at the top level of a script or module, it is in the **global namespace** (or global scope of that module).
* When you create a variable inside a function, it is in the **local namespace** of that function (local scope). It cannot be accessed outside that function.
* Python also has a built-in namespace for built-in functions and exceptions (like len, print, Exception, etc.).
* If Python encounters a name, it searches in the local scope first, then any enclosing scopes (if you have nested functions), then global scope, then built-in scope. This is called the LEGB rule (Local, Enclosing, Global, Built-in).

**Name collisions** occur when the same name is used in different namespaces or scopes. For example, a global variable x and a local variable x in a function are different, and the local one will shadow the global one within that function. Similarly, if you use a name that is the same as a built-in function, you might override it unintentionally.

**Example: Global vs Local**

x = 10 # global variable

def example():

x = 5 # local variable (this does not affect the global x)

print("Inside function, x =", x)

example()

print("Outside function, x =", x)

Output:

Inside function, x = 5

Outside function, x = 10

Inside the example() function, when we do x = 5, it creates a new local variable x. It doesn't change the global x. The print inside the function uses the local x. Outside, the global x is still 10.

If we wanted to modify the global x from inside the function, we'd have to declare it as global:

def example2():

global x # declare that we mean the global x

x = 7 # this will modify the global variable

print("Inside example2, x =", x)

example2()

print("Global x after example2 =", x)

Output:

Inside example2, x = 7

Global x after example2 = 7

Using global x tells Python that when we assign to x inside the function, it should go to the global variable.

**Example: Name collision with built-in:**

# Don't do this:

list = [1, 2, 3]

Here we created a variable named list, which is a built-in Python type for lists. Now list no longer refers to the list constructor. If we try to do list("abc"), it will fail because list is now a list object, not the function. To avoid such collisions, avoid naming your variables the same as built-in functions or types.

**Real-world tip:** If you have many variables and functions, organizing code into functions, classes, or modules can help avoid accidental name conflicts. Each function has its own local scope, and modules can have their own global variables that don’t interfere with other modules' globals.

**Exercise 14:** What will the following code print, and why?

value = 100

def change\_value(value):

value = 20

print("Inside function, value =", value)

change\_value(value)

print("Outside function, value =", value)

After understanding the output, how would you modify change\_value to actually change the global value? (Answer in words or code.)

**14. Importing Modules**

**Modules** in Python are files containing Python code (definitions and statements). A module can define functions, classes, and variables. Importing modules allows you to use code (functions, classes, variables) that others have written, including Python's vast standard library and external libraries.

Why use modules?

* **Code reuse:** You don't need to write everything from scratch. For example, Python has a math module for math functions, datetime for date and time, random for random number generation, etc.
* **Organization:** As your programs grow, you can split code into multiple files (modules) to keep it manageable. Each file is a module that can be imported.
* **Maintainability:** Using standard modules often means reliable and well-tested implementations (e.g., using json module to parse JSON rather than writing your own parser).

**Importing basics:**

* import module\_name – this imports the whole module. You access things from it with module\_name.some\_function.
* from module\_name import something – this imports a specific attribute (function, class, variable) from the module directly into your namespace, so you can use something without prefix.
* import module\_name as alias – this imports the module and gives it a shorter alias name (e.g., import numpy as np is common).

**Example: Importing and using modules**

import math

print(math.sqrt(16)) # Output: 4.0 (using the sqrt function from math module)

print(math.pi) # Output: 3.141592653589793 (using the constant pi)

from datetime import date

today = date.today()

print("Today's date:", today) # Uses date class from datetime module

import random as rnd

print(rnd.randint(1, 6)) # Output: a random integer between 1 and 6 (like a dice roll)

# You can also import your own modules (files) if you have them.

# For example, if you have a file helper.py with a function named do\_something,

# you could do: from helper import do\_something

In this example:

* We imported the entire math module and accessed sqrt and pi via the module name.
* We imported only the date class from datetime (so we didn't have to prefix it with datetime).
* We imported the random module with an alias rnd to save typing.
* We note that you can import your own files as modules (this requires that the file is in the same directory or in Python's path).

**Real-world scenario:** Modules are everywhere. If you need to fetch data from a URL, you'd import the requests library (after installing it). If you need to manipulate data tables, you might import pandas. Python's standard library has modules for JSON (json), regular expressions (re), file I/O (io), and many more. Importing is essential to leverage these.

**Exercise 15:** Write a small snippet that does the following:

1. Imports the math module.
2. Uses math.factorial to compute the factorial of 5 and prints the result.
3. Imports the random module (with alias r) and uses it to print a random floating-point number between 0 and 1 (hint: random.random() gives a float in [0.0, 1.0)).

**15. Comparison Operators and Basic Math Operations**

Python supports a variety of **comparison operators** and **arithmetic operators** to perform comparisons and calculations. We've used some of these in earlier examples, but here is a summary:

**Comparison (Relational) Operators:** These evaluate to True or False.

* == : equal to (checks if two values are equal).
* != : not equal to.
* > : greater than.
* < : less than.
* >= : greater than or equal to.
* <= : less than or equal to.

Examples:

5 == 5 # True

5 != 3 # True

7 > 10 # False

7 < 10 # True

2 >= 2 # True

3 <= 2 # False

**Basic Math (Arithmetic) Operators:**

* + : addition (also string concatenation for strings).
* - : subtraction.
* \* : multiplication.
* / : division (always returns a float in Python 3, e.g., 5/2 is 2.5).
* // : floor division (integer division that truncates the decimal, e.g., 5//2 is 2).
* % : modulus (remainder of division, e.g., 5 % 2 is 1).
* \*\* : exponentiation (power, e.g., 2 \*\* 3 is 8).
* +=, -=, etc.: These are shorthand for updating a variable. For example, x += 3 means x = x + 3.

Examples:

a = 10

b = 3

print(a + b) # 13

print(a - b) # 7

print(a \* b) # 30

print(a / b) # 3.3333333333333335

print(a // b) # 3 (because 10 // 3 = 3)

print(a % b) # 1 (remainder of 10/3)

print(a \*\* b) # 1000 (10 to the power of 3)

# Using +=

x = 5

x += 2 # equivalent to x = x + 2

print(x) # 7

Be mindful of the type when doing math:

* If one of the operands is a float, Python will convert the result to float (e.g., 3 + 2.0 yields 5.0).
* Division / always gives float. Use // if you need an integer result (floor division).

Also, Python respects **operator precedence** (order of operations). Multiplication, division, modulus, and exponentiation have higher precedence than addition and subtraction (and exponentiation \*\* has higher precedence than multiplication). You can use parentheses () to force a certain order.

**Exercise 16:** Evaluate the following expressions and write down the result (and type, if relevant) that each produces:

1. 5 + 2 \* 3
2. 5 + 2 \* 3 == 11
3. 8 / 3
4. 8 // 3
5. 8 % 3
6. 2 \*\* 3 \*\* 2 (hint: exponentiation is right-associative, so think of this as 2 \*\* (3 \*\* 2))
7. "Hello " + "World!"
8. "5" + "6"
9. "5" + 6 (what happens here?)

*(You can reason these out or quickly test them in a Python interpreter. For the last one, consider what type mismatch might occur.)*

**16. Error Handling with try and except**

Even the best programs will encounter errors at some point (like trying to open a file that doesn't exist, dividing by zero, etc.). Instead of crashing, you can **handle exceptions** (errors) gracefully using try and except blocks.

Basic syntax:

try:

# code that might raise an exception

risky\_operation()

except SomeErrorType:

# code to handle the error

recover\_or\_inform\_user()

except AnotherErrorType as e:

# handle a different type of error, with the error object as 'e'

print("Error:", e)

else:

# optional else block that runs if no exception was raised in try

print("Success!")

finally:

# optional finally block that runs no matter what (after try/except blocks)

cleanup()

For beginners, focus on try and except:

* Put code that might cause an error (like dividing, file I/O, type conversions, etc.) in the try block.
* Follow it with an except block to catch specific error types or a general Exception to catch anything.
* You can have multiple except blocks for different error types.
* Optionally use finally for cleanup actions (like closing a file) that must run regardless of success or failure.

**Example: Handling division by zero**

try:

numerator = 10

denominator = 0

result = numerator / denominator

print("Result:", result)

except ZeroDivisionError as e:

print("Cannot divide by zero! Error was:", e)

Output:

Cannot divide by zero! Error was: division by zero

In this example, the try block attempts a division by zero, which raises a ZeroDivisionError. The except catches that specific error and prints a friendly message instead of the program crashing with a traceback.

**Example: Handling invalid input conversion**

user\_input = "abc" # pretend this came from input()

try:

number = int(user\_input)

print("You entered number:", number)

except ValueError:

print(f"'{user\_input}' is not a valid integer.")

Output:

'abc' is not a valid integer.

Here, trying to convert "abc" to int raises a ValueError, which we catch and handle.

**Real-world scenario:** Use try/except to handle things like:

* Network requests (in case the connection fails, you can retry or show an error).
* File operations (if file not found, handle it).
* Parsing data (if format is wrong, handle the exception gracefully).

**Exercise 17:** Write a snippet of code that asks for two numbers and divides the first by the second. Use try and except to handle two potential errors:

1. The user might not enter a number (handle ValueError from int() conversion).
2. The user might enter 0 for the second number (handle ZeroDivisionError).

You can simulate the input by assigning values to variables for this exercise, or use actual input() calls if running interactively.

*(In the answer key, you can just show a simulated test with specific values, to avoid interactive input in this static format.)*

**17. Functional Programming vs Object-Oriented Programming in Python**

Python is a versatile language that supports multiple programming paradigms, mainly **object-oriented programming (OOP)** and **functional programming (FP)**. Understanding the difference can help you choose the right approach for a given problem.

* **Object-Oriented Programming (OOP):** This paradigm organizes code into objects (instances of classes). Each object contains data (attributes) and behavior (methods). OOP is useful for modeling complex systems with interacting objects. For example, in a banking application, you might have an Account class with methods like deposit and withdraw. OOP emphasizes concepts like encapsulation, inheritance, and polymorphism (though beginners can start with just understanding classes and objects).
* **Functional Programming (FP):** This paradigm treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. It emphasizes the use of functions as first-class citizens (meaning functions can be passed around just like data). FP in Python often means using pure functions (functions that don't have side effects), higher-order functions (functions that return or accept other functions), and operations like map, filter, reduce, list comprehensions, and generator expressions. Functional style code tends to be concise and focuses on "what to do" rather than "how to do it" step-by-step.

**Python lets you mix both styles.** You can write parts of your code in a functional style and others in an object-oriented style. In fact, many Python programs use a blend of both approaches.

Examples to illustrate:

* **Functional style example:** using functions and avoiding mutable state.
* # List of temperatures in Celsius, convert to Fahrenheit using map (functional approach)
* temps\_c = [0, 20, 35, 100]
* temps\_f = list(map(lambda c: (c \* 9/5) + 32, temps\_c))
* print(temps\_f) # Output: [32.0, 68.0, 95.0, 212.0]

Here we used map with a lambda to transform a list, rather than writing a loop with mutation.

* **Object-Oriented style example:** using classes and methods.
* class Temperature:
* def \_\_init\_\_(self, celsius):
* self.celsius = celsius
* def to\_fahrenheit(self):
* return (self.celsius \* 9/5) + 32
* temps = [Temperature(0), Temperature(20), Temperature(35), Temperature(100)]
* converted = [t.to\_fahrenheit() for t in temps]
* print(converted) # Output: [32.0, 68.0, 95.0, 212.0]

Here we modeled each temperature as an object of class Temperature and used a method to convert to Fahrenheit.

Both pieces of code achieve the same result (converting a list of Celsius temps to Fahrenheit). The first does it in a functional way (with a higher-order function map and a lambda expression), the second does it in an OO way (with a class and method, and a list comprehension to gather results).

**Which to use?**

* For simple transformations and data-focused tasks, functional style can be very concise and clear.
* For modeling complex data with multiple related behaviors, OOP can provide structure (for example, representing a user in a system with data and methods).
* Python's standard library and practices use both: e.g., we have built-in functions like len() (functional style global function), and string or list methods like .append() (OOP style method on an object).

In practice, don't worry too much about being strictly one or the other; use what makes the code easier to understand. Often you'll define classes for major components, and within methods you'll use functional tools like loops, list comprehensions, or map/filter as needed.

**Exercise 18:** For each scenario below, decide if it is more naturally suited for a functional approach or an object-oriented approach (or possibly both):

1. You have a list of numbers and you want to produce a new list with each number doubled.
2. You are creating a simulation of a library system with books, library members, and loans of books to members.

*(Justify your reasoning in one or two sentences for each part.)*

**Answer Key**

Below are the solutions or example answers to each exercise. Review these after you've attempted the exercises on your own.

**Exercise 1 Solution:** Properly formatted code with correct naming conventions:

amount = 100

def calc\_tax(amount):

rate = 0.1

tax = amount \* rate

print("Tax:", tax)

calc\_tax(amount)

Changes made:

* Renamed Amt to amount (lowercase, descriptive).
* Changed function name to calc\_tax (lowercase with underscore).
* Fixed indentation of the function definition (4 spaces for the body).
* Added spaces around = and \*.
* Lowercased Tax variable to tax.
* Ensured consistent naming and style.

**Exercise 2 Solution:**

months = 12 # int

temperature = 36.6 # float

message = "Hello" # str

colors = ["red", "green", "blue"] # list

book = {"title": "1984", "author": "George Orwell", "year": 1949} # dict

print(months)

print(temperature)

print(message)

print(colors)

print(book)

This creates the variables as specified and prints them. (The values can be different as long as the types are correct.)

**Exercise 3 Solution:**

* total\_cost – **Valid** (letters and underscore, starts with letter).
* 3d\_model – **Invalid** (starts with a digit).
* playerOne – **Valid** (letters only; case-sensitive but that's fine).
* def – **Invalid** (reserved keyword, used to define functions).
* email\_address – **Valid** (letters and underscore).
* first-name – **Invalid** (contains a hyphen).
* PI – **Valid** (letters only; by convention, all-caps might indicate a constant, but it's a valid name).

**Exercise 4 Solution:**

length\_str = "15"

width\_str = "8.4"

# Convert to appropriate types

length = int(length\_str) # "15" -> 15 (int)

width = float(width\_str) # "8.4" -> 8.4 (float)

area = length \* width

print("Area:", area) # Area: 126.0

We chose to convert length to int and width to float. The area is 126.0 (float). Converting both to float would also be fine (15.0 \* 8.4).

**Exercise 5 Solution:**

allowed\_users = ["alice", "bob", "charlie"]

email = "bob@example.com"

info = {"email": "bob@example.com", "age": 25, "member": True}

print("bob" in allowed\_users) # True

print("@" in email) # True

print("age" in info) # True

print(False in info.values()) # False, since False is not one of the values

Explanation:

1. "bob" in allowed\_users -> True (because "bob" is one of the list elements).
2. "@" in email -> True (the character "@" is present in the string).
3. "age" in info -> True (checks keys; "age" is a key in the dictionary).
4. False in info.values() -> False (the values in info are "bob@example.com", 25, and True. False is not among these).

**Exercise 6 Solution:**

data = [10, 20, 30, 40, 50, 60]

s = "abcdefg"

# 1. First three elements of data

print(data[:3]) # [10, 20, 30]

# 2. Last two elements of data

print(data[-2:]) # [50, 60]

# 3. Every other element of data

print(data[::2]) # [10, 30, 50] (elements at indices 0,2,4)

# 4. Substring "cde" from s

print(s[2:5]) # "cde" (chars at index 2,3,4)

# 5. Reverse the string s

print(s[::-1]) # "gfedcba"

These outputs match the intended slicing results.

**Exercise 7 Solution:**

temp\_c = -5

if temp\_c < 0:

print("Frozen (ice)")

elif temp\_c > 100:

print("Gas (steam)")

else:

print("Liquid water")

temp\_c = 25

if temp\_c < 0:

print("Frozen (ice)")

elif temp\_c > 100:

print("Gas (steam)")

else:

print("Liquid water")

temp\_c = 105

if temp\_c < 0:

print("Frozen (ice)")

elif temp\_c > 100:

print("Gas (steam)")

else:

print("Liquid water")

Expected outputs:

* For temp\_c = -5 -> Frozen (ice)
* For temp\_c = 25 -> Liquid water
* For temp\_c = 105 -> Gas (steam)

(Note: If temp\_c is exactly 0 or exactly 100, it falls into the else as "Liquid water" given our conditions.)

**Exercise 8 Solution (for loop sum):**

values = [3, 7, 1, 12, 9]

total = 0

for num in values:

total += num

print("Sum:", total) # Sum: 32

We initialize total to 0 and add each number. The sum of [3,7,1,12,9] is 32.

**Exercise 9 Solution (while loop 1 to 5):**

counter = 1

while counter <= 5:

print(counter)

counter += 1

Output:

1

2

3

4

5

This prints numbers 1 through 5 as required.

**Exercise 10 Solution:**

def square(x: (int, float)) -> (int, float):

# Optionally enforce type at runtime

if not isinstance(x, (int, float)):

print("Error: x must be a number (int or float)")

return None

return x \* x

# Testing the function

print(square(5)) # 25

print(square(2.5)) # 6.25

print(square("3")) # Error message, and returns None

Explanation:

* The function square returns the square of x. We've put a type hint that x can be int or float, and return can be int or float (in this simple case, if x is int -> returns int, if float -> returns float).
* The isinstance check will catch if someone calls square with a wrong type (like a string) and print an error message (and return None).
* The outputs should be 25, 6.25, and then an error message for the string case. (If copying this to run, the last call prints an error and returns None.)

**Exercise 11 Solution:**

class Book:

def \_\_init\_\_(self, title, author):

self.title = title

self.author = author

def description(self):

return f"\"{self.title}\" by {self.author}"

# Create an instance of Book

my\_book = Book("To Kill a Mockingbird", "Harper Lee")

# Call the description method

print(my\_book.description())

# Expected output: "To Kill a Mockingbird" by Harper Lee

We define the class with the required attributes and method, then create an instance and test the method.

**Exercise 12 Solution:**

1. After a[0] = 111:
   * a becomes [111, 200, [300, 400]].
   * b will also reflect this change, because b is just another reference to the same list. So b is [111, 200, [300, 400]].
   * c (the shallow copy) was a separate list initially, so its first element remains 100 (since that was copied by value for an int). After this operation, c is [100, 200, [300, 400]] (its own first element is still 100; it shares the inner list though).
2. After a[2].append(500):
   * a becomes [111, 200, [300, 400, 500]].
   * b, being the same list, is [111, 200, [300, 400, 500]] as well.
   * c (the shallow copy): its third element c[2] references the same inner list object as a[2]. So this inner list now has [300, 400, 500]. Thus c becomes [100, 200, [300, 400, 500]].

So in summary:

* After the first operation: a and b changed in the first element, c did not change in that position.
* After the second operation: all three (a, b, and c) show the added 500 in their inner list, because c's inner list was not a deep copy.

**Exercise 13 Solution:**

nums = [5, 8, 12, 15, 7, 1]

greater\_than\_10 = list(filter(lambda n: n > 10, nums))

print(greater\_than\_10) # Output: [12, 15]

This uses filter with a lambda to keep numbers > 10. The result is [12, 15] since those are the only numbers greater than 10 in the list.

**Exercise 14 Solution:** The code:

value = 100

def change\_value(value):

value = 20

print("Inside function, value =", value)

change\_value(value)

print("Outside function, value =", value)

will print:

Inside function, value = 20

Outside function, value = 100

Explanation: Inside the function, value is a local parameter (initially passed 100). We set this local value to 20, but this does not affect the global variable value. So outside the function, value is still 100.

To actually change the global value inside the function, you could:

* Use the global keyword:
* def change\_value\_global():
* global value
* value = 20

Then calling change\_value\_global() would set the global value to 20.

* Or return the new value and assign it:
* def change\_value(val):
* return 20
* value = change\_value(value)

This way, you reassign the global value with the returned result.

**Exercise 15 Solution:**

import math

print(math.factorial(5)) # Output: 120

import random as r

print(r.random()) # Output: (some random float between 0.0 and 1.0)

This imports math and uses math.factorial(5), which returns 120. Then it imports random with alias r and uses r.random() to print a random float.

**Exercise 16 Solution:**

1. 5 + 2 \* 3 -> first do 2 \* 3 = 6, then 5 + 6 = 11 (result is **11**, an int).
2. 5 + 2 \* 3 == 11 -> as above, left side is 11, so it's 11 == 11 -> **True** (a boolean).
3. 8 / 3 -> **2.6666666666666665** (a float, since division yields float).
4. 8 // 3 -> **2** (int, floor division).
5. 8 % 3 -> **2** (int, remainder of 8/3 since 3\*2=6 and remainder 2).
6. 2 \*\* 3 \*\* 2 -> This is 2 \*\* (3 \*\* 2) because exponentiation is right-associative. 3 \*\* 2 is 9, so it's 2 \*\* 9 -> **512**.
7. "Hello " + "World!" -> **"Hello World!"** (string concatenation).
8. "5" + "6" -> **"56"** (string concatenation, both are strings).
9. "5" + 6 -> This will cause a **TypeError** in Python, because you cannot directly add (concatenate) a string and an integer. You would need to convert 6 to a string or vice versa. (If we were to fix it, int("5") + 6 would be 11 or "5" + str(6) would be "56". But as given, it errors out.)

**Exercise 17 Solution:**

# Simulated input values:

num1\_str = "10" # user enters 10

num2\_str = "0" # user enters 0 (you can try "abc" for num2\_str to test ValueError case)

try:

a = int(num1\_str)

b = int(num2\_str)

result = a / b

print("Result:", result)

except ValueError:

print("One of the inputs is not a valid number.")

except ZeroDivisionError:

print("Cannot divide by zero.")

If num1\_str = "10" and num2\_str = "0", this will catch the ZeroDivisionError and print "Cannot divide by zero." If num2\_str was something like "abc", it would catch the ValueError when trying to convert to int. (If both errors could apply, the conversion happens first, so a non-numeric string would trigger the ValueError before trying division.)

**Exercise 18 Solution:**

1. Doubling a list of numbers – This can be done in either paradigm, but it is more naturally suited to a **functional approach**. You could use a simple loop (imperative style) or a list comprehension / map (functional style) to create a new list. There's no need for classes or objects here because it's a straightforward data transformation.
2. Library system simulation – This is more suited for an **object-oriented approach**. You would likely have classes like Book, Member, and maybe Loan to represent the entities and their interactions. Each book and member can be an object with attributes and methods (e.g., a Book object might have a title and author, a Member object might have a list of borrowed books, etc.). While you could manage it with just functions and data structures (functional/procedural style), using OOP would model the real-world entities more clearly in this scenario.