January 30, 2025

ITLPA701: PYTHON AND FUNDAMENTALS OF AI

LEARNING UNIT 2–DEVELOP PYTHON CONCEPT

Mr. Etienne NTAMBARA

ntambaraienne94@gmail.com

Assistant Lecturer in ICT Department

Rwanda Polytechnic, IPRC-HUYE

Homepage: https://94etienne.github.io/profile/

Learning hours: 9

IPRC HUYE

Learning Outcomes:

- 2.1. Writing Python syntax
- 2.3. Perform declaration
- 2.4. Defferentiate data type

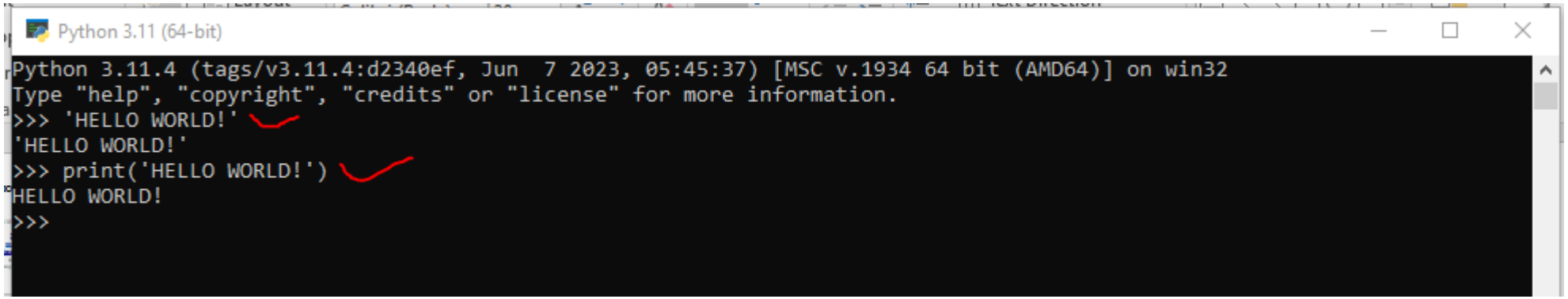
- Formative assessment 1: https://forms.gle/yrbyxPWweQqV9ufYA

Learning Outcome 2.1: Writing python syntax

- The Python syntax defines a set of rules that are used to create Python statements while writing a Python Program.
- Example :
- x = 5
- Age = 60
- Name = “MY NAME”
- x, y=8,9
- X = 7, Y = 10 => Wrong, float area = 9.0 => wrong
- The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages.
- However, there are some definite differences between the languages.
- Python extension is **.py**

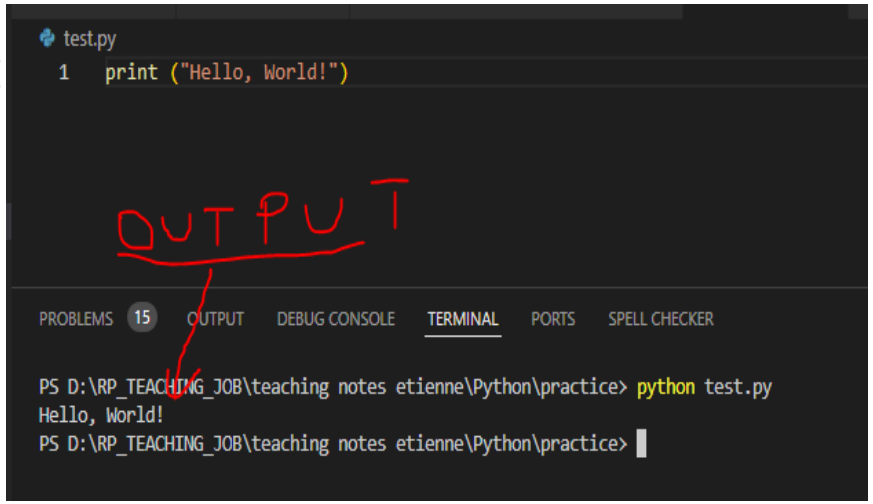
2.1.2. Use command line

- Let us execute a Python "**Hello, World!"** Programs in different modes of programming
- **2.1.2.1. Python modes**
- **1) Interactive Mode Programming**



2) Script Mode Programming

- We can invoke the Python interpreter with a script parameter which begins the execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.
- Let us write a simple Python program in a script which is simple text file. Python files have extension **.py**.
- Type the following source code in a test.py file:

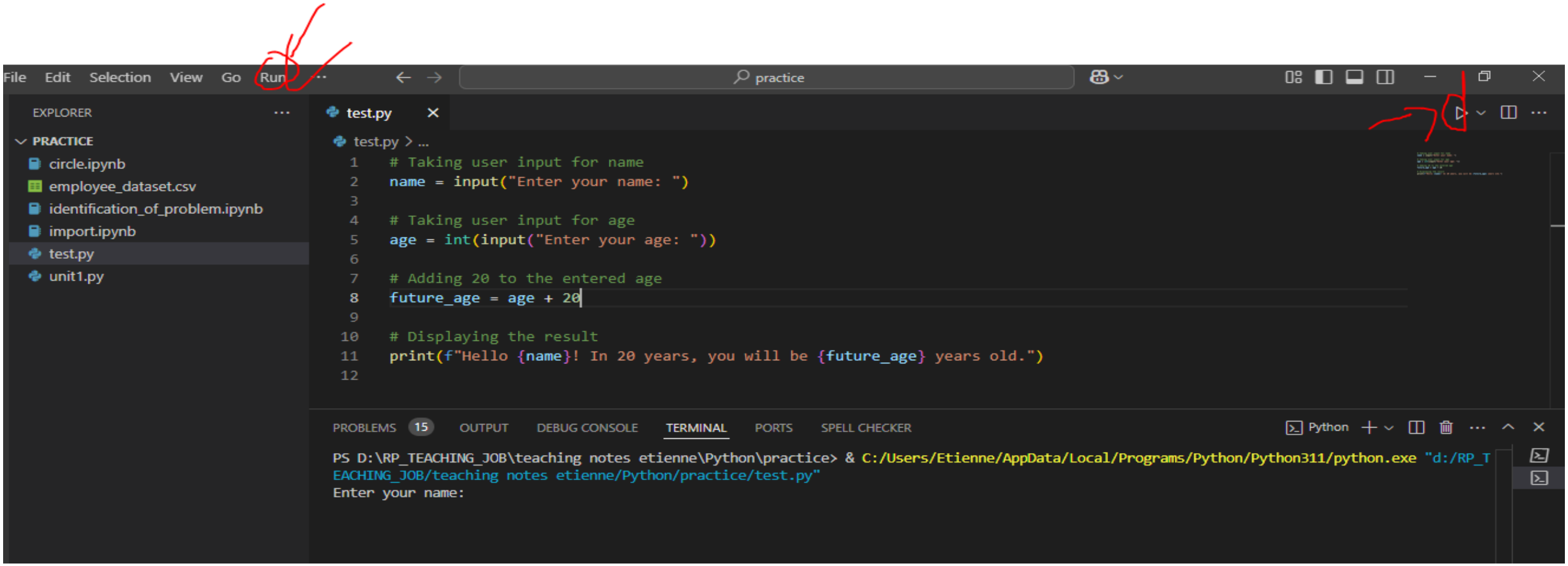


2.1.2.2 Compilation and execution

- In python compilation and execution are done whenever you run python program
- In terminal or command prompt: **Python3 simple_script.py** OR **Python simple_script.py**

2.1.2.2 Compilation and execution

- **In GUI (pycharm)**



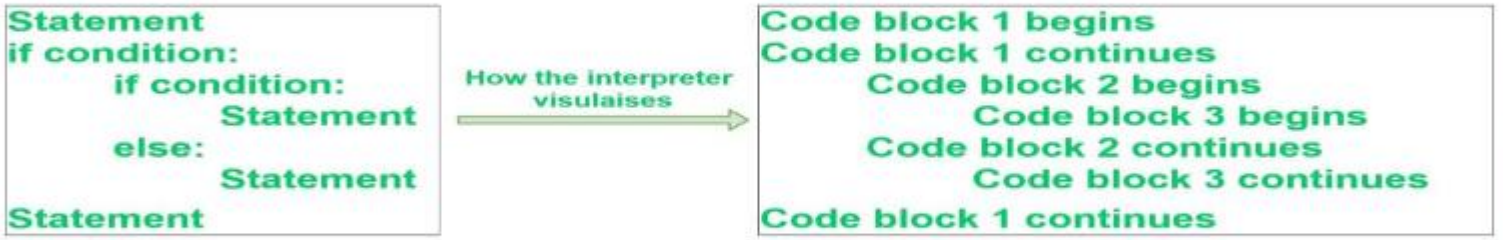
2.1.2.3 Save python project in VSCODE

- Go to file
- SAVE AS
- Or press ctrl + s

2.1.2.4 Indentation

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. **Python uses indentation to indicate a block of code**.

2.1.2.4 Indentation



Python programming provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by **line indentation**, which is rigidly enforced.
The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example:

```
if True:
  print ("True")
else:
  print ("False")
```

2.1.2.4 Indentation

- However, the following block generates an error:

```
if True:
print ("Answer")
print ("True")
else:
print ("Answer")
print ("False")
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

2.1.2.5 Python Reserved Words

2.1.2.5 Python Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	as	assert
break	class	continue
def	del	elif
else	except	False
finally	for	from
global	if	import
in	is	lambda
None	nonlocal	not
or	pass	raise
return	True	try
while	with	yield

2.1.3. Comments in Python

- Python comments are programmer-readable explanation or annotations in the Python source code.
- They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter.
- Comments enhance the readability of the code and help the programmers to understand the code very carefully.
- Just like most modern languages, Python supports **single-line** (or end-of-line) and **multi-line** (block) comments.
- Python comments are very much similar to the comments available in PHP, BASH and Perl Programming languages.

2.1.3. Comments in Python

- There are 2 types of comments available in Python
- Single line Comments
- Multiline Comments
- **2.1.3.1. Single Line Comments**
- A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them
- Following is an example of a single line comment in Python: **#**
- # This is a single line comment in python
- print ("Hello, World!")
- This produces the following result: Hello, World!

2.1.3. Comments in Python

- You can type a comment on the same line after a statement or expression.
- name = "Madisetti" # This is again comment
- 2.1.3.2. Multi-Line Comments
- Python does not provide a direct way to comment multiple line. You can comment multiple lines as follows:
- Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

```
"""
Taking user input for name
Taking user input for age
Adding 20 to the entered age
Displaying the result
"""
```

Learning Outcome 2.2: Perform declaration

2.2.1. Definition of Key terms

- Declaration
- Variables

2.2.1.1 Variable declaration and Assignment

- Python variables are the reserved memory locations used to store values with in a Python Program.
- This means that when you create a variable you reserve some space in the memory.
- Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory.
- Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.

2.2.1.2. Variable declaration

- Python variables do not need explicit declaration to reserve memory space or you can say to create a variable.
- A Python variable is created automatically when you assign a value to it. The equal sign **(=)** is used to assign values to variables.

2.2.2. Assigning values

- Single value
- Multiple values

2.2.2.1. Single value assignment

- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example:

```python
# Assigning values to variables
x = 5          # 'x' is the variable name, and 5 is the value assigned to it
name = "John"  # 'name' is the variable name, and "John" is the value assigned to it

# Printing variables to check their values
print(x)       # Output: 5
print(name)    # Output: John
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python assigning_values.py
5
John
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.2.2. Multiple value assignment

- Multiple value assignment in Python allows you to assign values to multiple variables in a single line. This feature simplifies code and makes assignments more concise and readable.
- Example 1: Assigning Values to Multiple Variables

multiple_variables.py > ...

```python
# Assigning values to multiple variables in one line
x, y, z = 10, 20, 30

print(x)  # Output: 10
print(y)  # Output: 20
print(z)  # Output: 30
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python multiple_variables.py
10
20
30
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.2.2. Multiple value assignment

- Python allows you to assign a single value to several variables simultaneously which means you can create multiple variables at a time.
- Example 2: Assigning the Same Value to Multiple Variables:

```python
# Assigning the same value to multiple variables
a = b = c = 100

print(a)  # Output: 100
print(b)  # Output: 100
print(c)  # Output: 100
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python multiple_variables.py
100
100
100
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.2.2. Multiple value assignment

- Example 3: Swapping Variables:

```python
# Swapping values without using a temporary variable
x, y = 5, 10
x, y = y, x

print(x)  # Output: 10
print(y)  # Output: 5
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python multiple_variables.py
10
5
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.2.2. Multiple value assignment

- Example 4: Using Unpacking with Lists or Tuples:

```python
# Assigning list or tuple elements to variables
values = (1, 2, 3)
a, b, c = values

print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python multiple_variables.py
1
2
3
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.2.2. Multiple value assignment

- Example 5: Ignoring Certain Values:

```python
# Using underscore (_) to ignore a value during unpacking
x, _, z = (10, 20, 30)

print(x)  # Output: 10
print(z)  # Output: 30
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python multiple_variables.py
10
30
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Above examples shows the different **ways or methods** of handling multiple value assignments in Python.

1. Assigning Different Values to Multiple Variables:

- Example: x, y, z = 10, 20, 30
- This assigns individual values to multiple variables in one line.

2. Assigning the Same Value to Multiple Variables:

- Example: a = b = c = 100
- This assigns the same value to all the specified variables.

3. Swapping Variables:

- Example: x, y = y, x
- This swaps the values of two variables without needing a temporary variable.

4. Unpacking Values from Data Structures (Lists, Tuples):

- Example: a, b, c = (1, 2, 3)
- Assigns elements of a tuple or list to variables directly.

6. Ignoring Specific Values Using _:

- Example: x, _, z = (10, 20, 30)
- The underscore _ is used to ignore certain values during unpacking.

2.2.1.2. Python Variable Names

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like

color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

i. A variable name must start with a letter or the underscore character

ii. A variable name cannot start with a number or any special character like $, (, * % etc.

iii. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

iv. Python variable names are case-sensitive which means Name and NAME are two different variables in Python.

v. Python reserved keywords cannot be used naming the variable.

Examples of Valid Variable Names

valid_variable_names.py > ...

```python
# Starts with a letter
name = "John"
age = 25
# Starts with an underscore
_person = "Alice"
# Contains alphanumeric characters and underscores
employee_id = 101
total_score_2025 = 95
print(name)
print(age)
print(_person)
print(employee_id)
print(total_score_2025)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python valid_variable_names.py
John
25
Alice
101
95
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Examples of Invalid Variable Names

invalid_variable_names.py > ...

```python
# Variable name cannot start with a number
1name = "Error"  # SyntaxError

# Variable name cannot start with a special character
$name = "Error"  # SyntaxError

# Variable name cannot contain special characters other than underscores
first-name = "Error"  # SyntaxError

# Python reserved keywords cannot be used
class = "Error"  # SyntaxError
```

Examples of Case-Sensitive Variable Names

case_sensintive.py > ...

```python
Name = "Alice"
name = "Bob"

print(Name)  # Output: Alice
print(name)  # Output: Bob
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python case_sensintive.py
Alice
Bob
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Key Takeaways:

- Always start a variable name with a letter or underscore.
- Avoid starting variable names with numbers or special characters.
- Use only alphanumeric characters and underscores in variable names.
- Remember that Python is case-sensitive.
- Avoid reserved keywords as variable names

2.2.3. Types of variables

- Local
- Global
- Global keywords

2.2.3.1. Python Local Variable

- Python Local Variables are defined inside a function. We can not access variable outside the function.
- Example 1:

local_variable.py > ...

```python
def sum(a,b):
    sum = a+b
    return sum
a = int(input("Enter num 1:"))
b = int(input("Enter num 1:"))
result = sum(a,b)
print(f"Sum of {a} and {b} is {result}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python local_variable.py
Enter num 1: 5
Enter num 1:6
Sum of 5 and 6 is 11
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.3.1. Python Local Variable

- Example 2

local_variable.py > ...

```python
# Function to calculate the area of a rectangle
def compute_area(length, width):
    # 'area' is a local variable defined inside this function.
    # It is created when the function is called and is destroyed when the function finishes execution.
    area = length * width  # Formula for area of a rectangle
    return area  # The local variable 'area' is returned to the caller

# Input: Length and width of the rectangle
# These variables are global since they are defined outside any function.
length = float(input("Enter the length of the rectangle: "))
width = float(input("Enter the width of the rectangle: "))

# Compute the area
# Here, the values of 'length' and 'width' are passed to the function.
area = compute_area(length, width)

# Output: Display the area
# The 'area' variable is a global variable that stores the result returned by the function.
print(f"The area of the rectangle is: {area} square units")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python local_variable.py
Enter the length of the rectangle: 30
Enter the width of the rectangle: 3
The area of the rectangle is: 90.0 square units
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Key Points About Local Variables

- A local variable is created when the function is called and destroyed once the function execution is completed.
- It cannot be accessed or modified outside its defining function.
- Local variables are useful for temporary or intermediate calculations within a function.

2.2.3.2. Python Global Variable

- Any variable created outside a function can be accessed within any function and so they have global scope.
- Following is an example of global variables:

global_variable.py > ...

```python
# Global variable x initialized with a value of 10
x = 10
# Global variable y initialized with a value of 20
y = 20
# Function to calculate the sum of the global variables x and y
def sum():
    # Accessing global variables x and y
    sum = x + y
    return sum

# Print the result of the sum function
# sum() => Function call
print(f"Sum of {x} and {y} is = {sum()}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python global_variable.py
Sum of 10 and 20 is = 30
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.2.3.2 Global keywords

- The global keyword in Python is used to modify a variable outside the current function's scope, allowing functions to update global variables.

```python
x = 10  # Global variable

def update_x():
    global x  # Declare x as global
    x = 20  # Modify global x

update_x()
print(x)  # Output: 20
```

Key Points:

- Without `global`, `x` inside the function would be treated as a local variable.
- Using `global` allows functions to modify variables defined outside their local scope.

Learning Outcome 2.3: Defferentiate data type

- Python Data Types are used to define the type of a variable.
- It defines what type of data we are going to store in a variable. The data stored in memory can be of many types.
- For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters

2.3.1. Define build-in data type

- Python has various built-in data types:

1. Text
2. Numeric
3. Sequence
4. Mapping
5. String
6. Booleans

2.3.1.1. Text

- This includes **strings** in Python, which are used to store textual data. Strings can be enclosed in **single**, **double**, or **triple** quotes.

string1.py > ...

```python
text = "Hello, Python!"  # Text data
print(text)
print(f"My Data type is: {type(text)}")  # Output: <class 'str'>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python string1.py
Hello, Python!
My Data type is: <class 'str'>
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.3.1.2. Numeric

Numeric data types represent numbers. Python has three main numeric types:

- **int**: Integer values (e.g., 10, -5)
- **float**: Decimal numbers (e.g., 10.5, -0.99)
- **complex**: Numbers with a real and imaginary part (e.g., 2+3j)

numeric.py > ...

```python
integer_num = 42          # int
float_num = 3.14          # float
complex_num = 1 + 2j      # complex

print(type(integer_num))  # Output: <class 'int'>
print(type(float_num))    # Output: <class 'float'>
print(type(complex_num))  # Output: <class 'complex'>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python numeric.py
<class 'int'>
<class 'float'>
<class 'complex'>
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.3.1.3. Sequence

Sequence types are used to store multiple values in an ordered manner. Common sequence types include:

- **list**: Mutable, ordered collection of items.
- **tuple**: Immutable, ordered collection of items.
- **range**: Sequence of numbers generated lazily.

2.3.1.3. Sequence

Example

sequence.py > ...

```python
print("----" * 10)
# List
my_list = [1, 2, 3]
print("List:",my_list)
print(type(my_list))  # Output: <class 'list'>
print("----" * 10)
# Tuple
my_tuple = (1, 2, 3)
print("Tuple:",my_tuple)
print(type(my_tuple))  # Output: <class 'tuple'>
print("----" * 10)
# Range
my_range = range(1, 5)  # 1, 2, 3, 4
print("Range:",my_range)
print(type(my_range))  # Output: <class 'range'>
print("----" * 10)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python sequence.py
----------
List: [1, 2, 3]
<class 'list'>
----------
Tuple: (1, 2, 3)
<class 'tuple'>
----------
Range: range(1, 5)
<class 'range'>
----------
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.3.1.4. Mapping

A mapping type in Python is a collection of key-value pairs. The most common mapping type is a **dictionary**.

Example:

mapping.py > ...

```python
my_dict = {"name": "Alice", "age": 25}  # Dictionary
print("Mapping: ",my_dict)
print(type(my_dict))  # Output: <class 'dict'>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python mapping.py
Mapping:  {'name': 'Alice', 'age': 25}
<class 'dict'>
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.3.1.5. String

A **String** is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it. Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text. Python Strings are identified as a contiguous set of characters represented in the quotation marks. **Python allows for either pairs of single or double quotes.**

Creating a String in Python

Strings in Python can be created using single quotes or double quotes or even triple quotes. Let us see how

we can define a string in Python.

2.3.1.5. String

Example: In this example, we will demonstrate different ways to create a Python String. We will create a string using single quotes **(‘ ‘),** double quotes **(” “),** and triple double quotes **(“”” “”””)**. The triple quotes can be used to declare multiline strings in Python

string2.py > ...

```python
# String using single quotes
string1 = 'Hello, Python!'
print(string1)

# String using double quotes
string2 = "Hello, Python!"
print(string2)

# Multiline string using triple quotes
string3 = """Hello,
This is a multiline string in Python!"""
print(string3)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python string2.py
Hello, Python!
Hello, Python!
Hello,
This is a multiline string in Python!
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

String Operations: Insert, Update, and Delete in Python

Python strings are **immutable**, meaning their content cannot be directly modified. However, you can achieve operations like **insert**, **update**, and **delete** by creating a new string based on the desired changes. Below are examples demonstrating these operations.

String Operations: Insert, Update, and Delete in Python

Python strings are **immutable**, meaning their content cannot be directly modified. However, you can achieve operations like **insert**, **update**, and **delete** by creating a new string based on the desired changes.

Below are examples demonstrating these operations.

Inserting a Substring

To insert a substring at a specific position in a string:

original[:position]:

This slices the string from the beginning up to (but not including) the index position.

For example, if original = "Hello Python!" and position = 6, original[:6] gives "Hello ".

"World ":

This is the string you are inserting. It will be added between the slices of the original string.

original[position:]:

This slices the string from the index position to the end of the string.

Using the same example, original[6:] gives "Python!".

insert_string.py > ...

```python
# Original string
original = "Hello Python!"
# Insert "World " after "Hello "
position = 6
new_string = original[:position] + "World " + original[position:]
print(new_string)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python insert_string.py
Hello World Python!
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Updating a String

To update part of a string, replace the desired portion:

update_string.py > ...

```python
original = "Hello World!"
# Replace "World" with "Python"
updated_string = original.replace("World", "Python")
print(updated_string)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python update_string.py
Hello Python!
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Deleting a Substring

- To delete a substring, remove it by slicing:

delete_string.py > ...

```python
original = "Hello World!"
# Remove "World"
substring_to_remove = "World"
new_string = original.replace(substring_to_remove, "")
# Strip extra spaces if necessary
new_string = new_string.strip()
print(new_string)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python delete_string.py
Hello !
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2.3.1.6. Booleans

- Python **boolean** type is one of built-in data types which represents one of the two values either **True** or **False**.
- Python **bool()** function allows you to evaluate the value of any expression and returns either **True** or **False** based on the expression.

Examples of Boolean Usage:

1. Evaluating Numbers:
2. Evaluating Strings:
3. Evaluating Lists, Tuples, and Other Containers:
4. Evaluating Logical Expressions

1. Evaluating Numbers

boolean_numbers.py

```python
# Non-zero numbers evaluate to True
print(bool(1))  # Output: True
print(bool(-5))  # Output: True

# Zero evaluates to False
print(bool(0))  # Output: False

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python boolean_numbers.py
True
True
False
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

2. Evaluating Strings

boolean_string.py

```python
# Non-empty strings evaluate to True
print(bool("Hello"))  # Output: True
print(bool("Python"))  # Output: True

# Empty strings evaluate to False
print(bool(""))  # Output: False
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python boolean_string.py
True
True
False
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

3. Evaluating Lists, Tuples, and Other Containers:

boolean_tuples.py

```python
# Non-empty containers evaluate to True
print(bool([1, 2, 3]))  # Output: True
print(bool({"key": "value"}))  # Output: True

# Empty containers evaluate to False
print(bool([]))  # Output: False
print(bool(()))  # Output: False
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python boolean_tuples.py
True
True
False
False
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

4. Evaluating Logical Expressions:

boolean_logical.py

```python
# Comparisons return boolean values
print(10 > 5)  # Output: True
print(3 == 4)  # Output: False

# Combining boolean values with logical operators
print(True and False)  # Output: False
print(True or False)   # Output: True

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2> python boolean_logical.py
True
False
False
True
PS D:\RP_TEACHING_JOB\teaching notes etienne\Python\practice\unit2>
```

Conclusion

- The boolean type is a fundamental part of Python, used in various operations and decision-making processes. The bool() function can help determine whether a value or expression is logically True or False

2.3.2. Numbers

Python provides three main numeric data types to handle different kinds of numbers:

- Integer (**int**)
- Float (**float**)
- Complex (**complex**)

2.3.2.1. Integer (int)

Definition: Represents whole numbers (positive, negative, or zero) without any fractional or decimal component.

- **Characteristics**:
- No decimal point.
- Can be arbitrarily large in Python (unlike fixed-size integers in some other languages).

Examples:

```python
# Examples of integers
a = 10   # Positive integer
b = -20  # Negative integer
c = 0    # Zero
print(type(a))  # Output: <class 'int'>
```

2.3.2.2. Float (float)

Definition: Represents real numbers that contain a decimal point. Used for precise calculations with fractions.

- **Characteristics**:
- Includes numbers with a decimal point or written in scientific notation.

- Characteristics:
  - Includes numbers with a decimal point or written in scientific notation.
  - Example of scientific notation:

python Copy Edit

```python
num = 1.23e4  # Equivalent to 1.23 * 10^4
print(num)    # Output: 12300.0
```

Examples:

python

```python
# Examples of floats
x = 10.5     # Positive float
y = -3.14    # Negative float
z = 0.0      # Zero as a float
print(type(x))  # Output: <class 'float'>
```

2.3.2.3. Complex (complex)

Definition: Represents complex numbers, which consist of a real part and an imaginary part. The imaginary part is denoted with a **j**.

Characteristics:

- Real and imaginary parts are stored as floats.
- You can access the real and imaginary parts using .real and .imag attributes.

Characteristics:

- Real and imaginary parts are stored as floats.
- You can access the real and imaginary parts using `.real` and `.imag` attributes.

python Copy Edit

```python
comp = 5 + 6j
print(comp.real)  # Output: 5.0
print(comp.imag)  # Output: 6.0
```

Examples:

python

```python
# Examples of complex numbers
comp1 = 2 + 3j   # Real part: 2, Imaginary part: 3
comp2 = -1 - 4j  # Real part: -1, Imaginary part: -4
print(type(comp1))  # Output: <class 'complex'>
```

Summary of Numeric Data Types

Summary of Numeric Data Types

Data Type	Example Values	Use Case
`int`	`10, -5, 0`	Counting or indexing without decimals.
`float`	`3.14, -2.5, 0.0`	Precise calculations with fractional parts.
`complex`	`2+3j, -1-4j`	Mathematical operations with imaginary numbers.

Example Demonstration in Python:

```
# Integer
a = 42
print(f"{a} is of type {type(a)}")  # Output: 42 is of type <class 'int'>

# Float
b = 3.14159
print(f"{b} is of type {type(b)}")  # Output: 3.14159 is of type <class 'float'>

# Complex
c = 2 + 5j
print(f"{c} is of type {type(c)}")  # Output: (2+5j) is of type <class 'complex'>
print(f"Real part: {c.real}, Imaginary part: {c.imag}")  # Output: Real part: 2.0, Imaginary part: 5.0
```

Data types summary

Data Types Examples

`x = apple`	`string`
`x = 3.14`	`float`
`x = 1j`	`complex`
`x = True, False`	`bool`
`x = ["apple", "banana", "cherry"]`	`list`
`x = ("apple", "banana", "cherry")`	`tuple`
`x = range(6)`	`range`
`x = {"name" : "John", "age" : 36}`	`dict`
`x = {"apple", "banana", "cherry"}`	`set`
`x = frozenset({"apple", "banana", "cherry"})`	`frozenset`

Formative Assessment 2

Practical Assignment

https://drive.google.com/file/d/1RPdmUZMHPBZmZAK9ecQsWmAjLBQkaEWe/view?usp=sharing