

INTRODUCTION TO GIT CONTIN.

git pull is a Git command that **updates your local copy of a repository** with the latest changes from a remote repository (like GitHub).

It is actually a combination of two commands:

1. **git fetch** – downloads the latest commits from the remote repository but **doesn't change your files yet**.
2. **git merge** – merges the fetched changes into your current branch automatically.

So when you run **git pull**, Git fetches the latest changes and tries to merge them into the branch you are currently working on.

How It Interacts with Branches

- Git tracks **branches** separately. Each branch can have its own set of commits.
- When you do **git pull**, it **only updates the branch you are currently on**.
- If the remote branch has new commits, Git will bring them into your local branch.
- If your local branch has changes that conflict with the remote branch, Git will **ask you to resolve conflicts** before finishing the merge.

Lets see how to use **git pull** to grab the latest changes for our repo so you can get the latest changes & updates before each class:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER
(venv) doriannins@Dorians-MacBook-Studio _Coding_Course % clear
/Users/doriannins/Library/CloudStorage/Dropbox/Laghima_Videos/_Coding_Course
(venv) doriannins@Dorians-MacBook-Studio _Coding_Course % ls
CodingMentorshipWhopCodeBase
(venv) doriannins@Dorians-MacBook-Studio _Coding_Course % cd CodingMentorshipWhopCodeBase
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % LS
HW      NOTES      README.md
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % ls HW
CC_WK1_HW.pdf  CC_WK2_HW.pdf  README.md
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % ls NOTES
CC_WK1.pdf     CC_WK2.pdf
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % git branch
* main
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase %
```

```
PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER
(venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % clear
* main
• (venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % git remote -v
origin https://github.com/laghimamusic/CodingMentorshipWhopCodeBase.git (fetch)
origin https://github.com/laghimamusic/CodingMentorshipWhopCodeBase.git (push)
• (venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % git pull
Already up to date.
• (venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % ls hw
CC_WK1_HW.pdf CC_WK2_HW.pdf CC_WK3_HW.py README.md
• (venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase % ls NOTES
CC_WK1.pdf CC_WK2.pdf CC_WK3.pdf
○ (venv) doriannins@Dorians-MacBook-Studio CodingMentorshipWhopCodeBase %
```

WHAT IS PYTHON?

Python programming is an accessible, high-level programming language designed for simplicity and readability, making it ideal for beginners and experts alike. It allows you to write instructions that a computer can execute, bridging the gap between human language and machine code. Unlike binary code, which computers inherently understand but humans find complex, Python uses a syntax closer to English, thus making programming more intuitive.

Python differs from traditional programming languages like C in several fundamental ways, making it more accessible and user-friendly, especially for beginners. Firstly, Python is an interpreted language, meaning that the Python interpreter directly executes the code line-by-line, allowing for immediate feedback and easier debugging. In contrast, C is a compiled language where the code must be transformed into machine code through a compilation process before execution, which can be more complex and time-consuming.

Python emphasizes simplicity and readability with a syntax that is closer to natural English, reducing the need for verbose code and complex syntax rules that are common in C. For instance, Python manages memory automatically through garbage collection, whereas C requires manual memory management, which increases the risk of errors. Python abstracts many low-level details like variable declarations and pointer management, which are essential in C but can be daunting for beginners.

Moreover, Python supports dynamic typing, meaning variables do not require explicit type declarations, and their types can change at runtime. C, on the other hand, uses static typing, requiring explicit type definitions, which enforces stricter control but also adds complexity. Python's extensive standard library and built-in functions, such as the `print()` function for output, simplify programming tasks, whereas C often requires more code and external libraries to achieve similar functions.

PYTHON SYNTAX

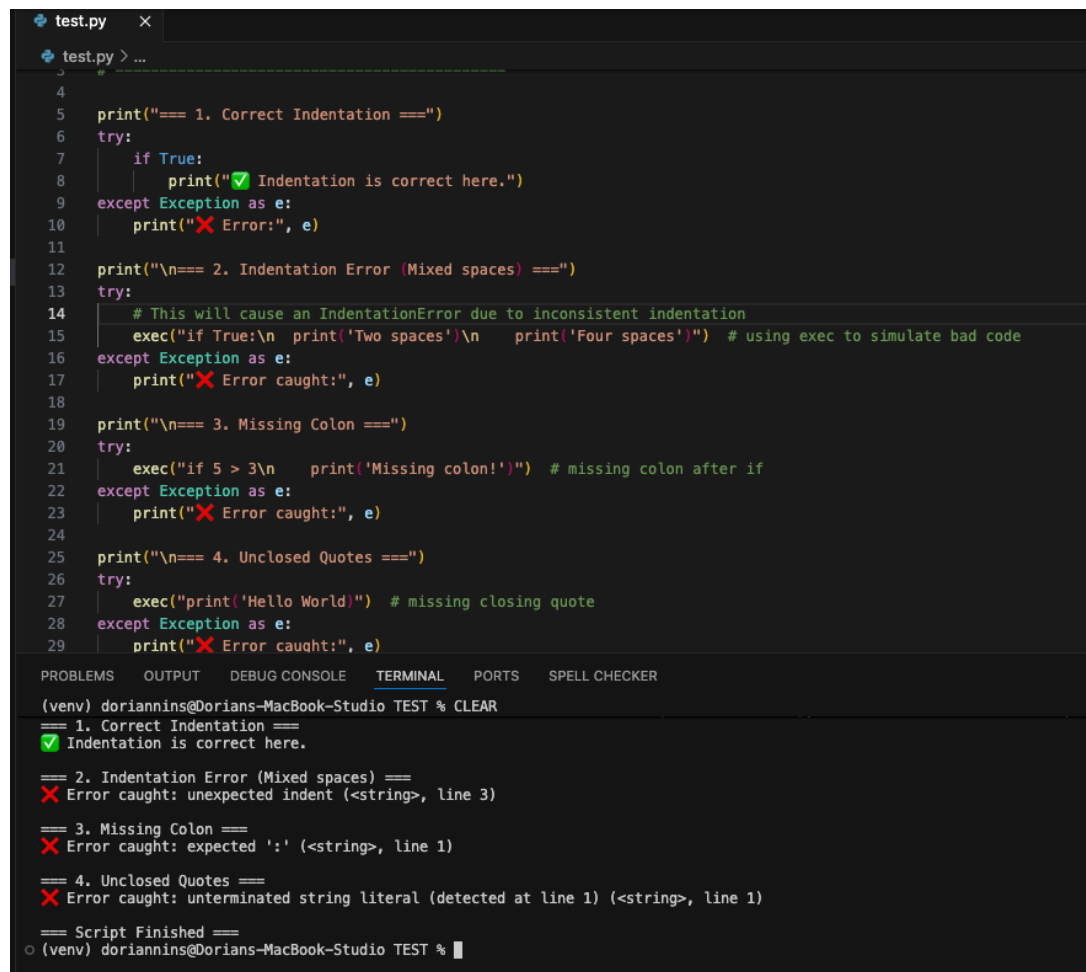
Syntax is just the set of rules for writing Python so the computer can understand what you mean. It's kind of like spelling and punctuation in English. If you leave out a period or mix up letters, the sentence stops making sense.

In Python, things like using the right spacing, adding a colon after an if statement, or closing your quotes are all part of syntax.

If you don't follow those rules, Python gets confused and throws an error instead of running your code.

Good syntax makes your code easy to read for you and clear for the computer to follow.

Lets look at an example:



```
test.py x
test.py > ...
3 # =====
4
5 print("=== 1. Correct Indentation ===")
6 try:
7     if True:
8         print("✅ Indentation is correct here.")
9 except Exception as e:
10    print("❌ Error:", e)
11
12 print("\n=== 2. Indentation Error (Mixed spaces) ===")
13 try:
14    # This will cause an IndentationError due to inconsistent indentation
15    exec("if True:\n    print('Two spaces')\n    print('Four spaces')") # using exec to simulate bad code
16 except Exception as e:
17    print("❌ Error caught:", e)
18
19 print("\n=== 3. Missing Colon ===")
20 try:
21    exec("if 5 > 3\n    print('Missing colon!')") # missing colon after if
22 except Exception as e:
23    print("❌ Error caught:", e)
24
25 print("\n=== 4. Unclosed Quotes ===")
26 try:
27    exec("print('Hello World')") # missing closing quote
28 except Exception as e:
29    print("❌ Error caught:", e)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER
(venv) doriannins@Dorians-MacBook-Studio TEST % CLEAR
=== 1. Correct Indentation ===
✅ Indentation is correct here.

=== 2. Indentation Error (Mixed spaces) ===
❌ Error caught: unexpected indent (<string>, line 3)

=== 3. Missing Colon ===
❌ Error caught: expected ':' (<string>, line 1)

=== 4. Unclosed Quotes ===
❌ Error caught: unterminated string literal (detected at line 1) (<string>, line 1)

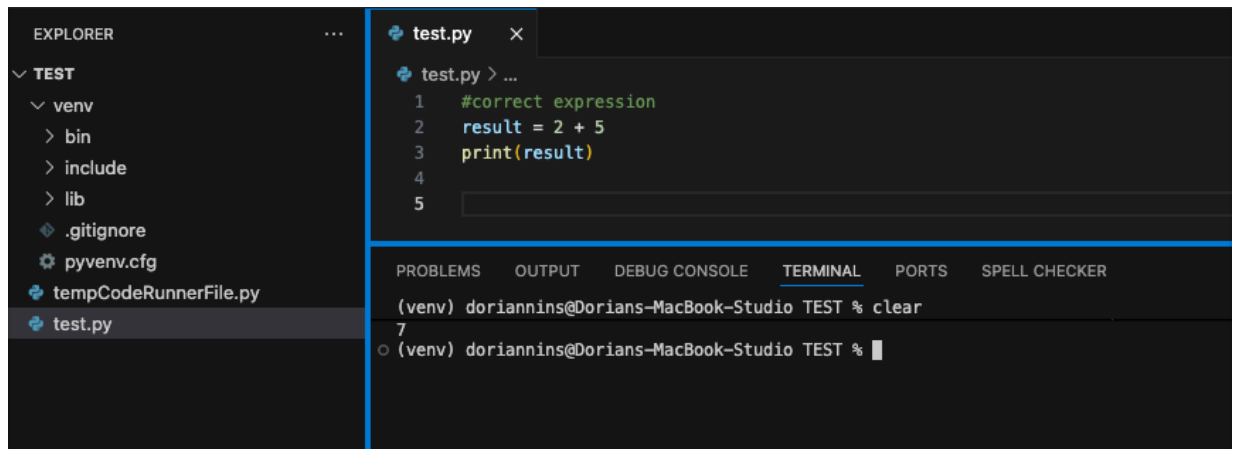
=== Script Finished ===
o (venv) doriannins@Dorians-MacBook-Studio TEST %
```

NUMERIC SYNTAX

When writing Python code, understanding some generic syntax rules is essential to ensure your code runs smoothly and avoids common errors. Python syntax is designed to be simple and readable, but it still requires strict adherence to grammar rules.

First, all statements must follow correct syntax. For example, if you perform arithmetic operations like addition or subtraction, you write $2 + 3$ or $9 - 8$. Python automatically recognizes integers and floats, but if you miss part of an expression, such as writing $8 + 9 -$ without a number afterward, you will get a syntax error like invalid syntax. This happens because Python expects a complete expression.

Lets look at an correct example:



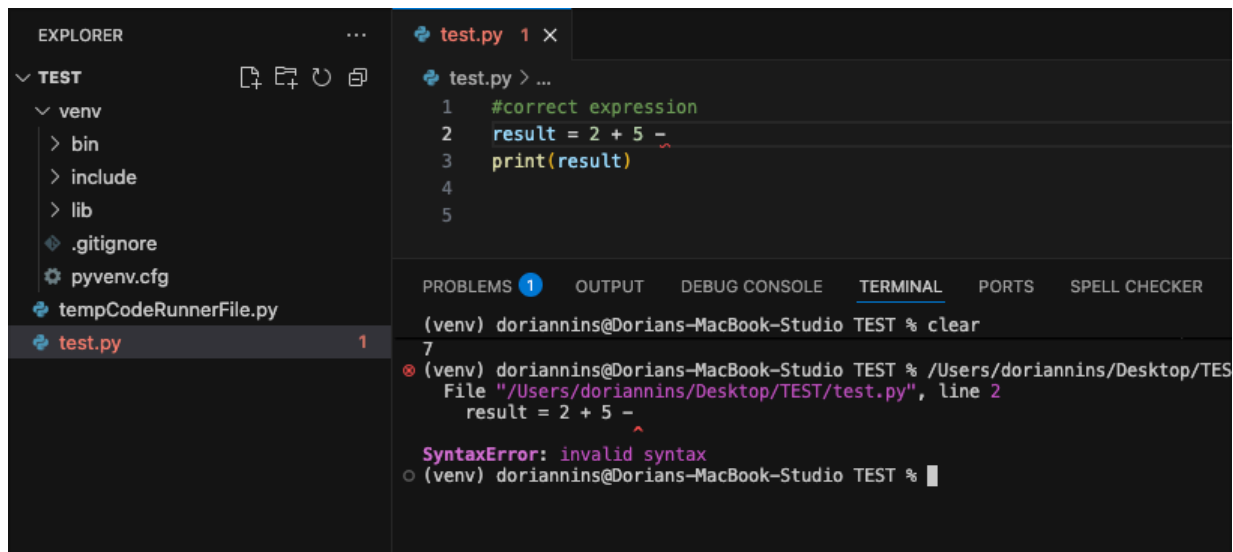
The screenshot shows the VS Code interface with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'TEST' with a subdirectory 'venv'. The code editor shows a file named 'test.py' with the following content:

```
test.py > ...
1 #correct expression
2 result = 2 + 5
3 print(result)
4
5
```

The terminal output shows the command `(venv) doriannins@Dorians-MacBook-Studio TEST % clear` and the output `7`.

As you can see this correct syntax results in a proper numerical output

Lets look at an incorrect example:



The screenshot shows the VS Code interface with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'TEST' with a subdirectory 'venv'. The code editor shows a file named 'test.py' with the following content:

```
test.py 1 X
test.py > ...
1 #correct expression
2 result = 2 + 5 -
3 print(result)
4
5
```

The terminal output shows the command `(venv) doriannins@Dorians-MacBook-Studio TEST % clear` and the output `7`. Below the output, a syntax error is shown:

```
File "/Users/doriannins/Desktop/TEST/test.py", line 2
    result = 2 + 5 -
                  ^
SyntaxError: invalid syntax
```

Notice how the expression is not complete there is a linger subtraction sign also notice how VS Codes Python interpreter has underlined this portion with a red line that means running this code will result in a error as we can see it does in the command line output

NUMERIC OPERATORS?

In Python programming, several numeric operators allow you to perform arithmetic operations on numbers, which include both integers and floating-point values.

These operators form the foundation of mathematical computations in your code, and understanding them is essential.

1. **Addition (+)**: This operator adds two numbers.
Example: $2 + 3$ results in 5.
2. **Subtraction (-)**: This subtracts the right operand from the left operand.
Example: $9 - 8$ results in 1.
3. **Multiplication (*)**: Multiplies two numbers.
Example: $4 * 6$ results in 24.
4. **Division (/)**: Divides the left operand by the right operand and returns a floating-point result (float).
Example: $8 / 4$ results in 2.0. Even if the division is exact, the output is a float.
5. **Integer Division (//)**: Divides the left operand by the right operand and returns the quotient without the decimal part (an integer).
Example: $5 // 2$ results in 2.
6. **Modulus (%)**: Returns the remainder of the division between two numbers.
Example: $10 \% 3$ results in 1 because 10 divided by 3 leaves a remainder of 1.
7. **Exponentiation (**)**: Raises the left operand to the power of the right operand.
Example: $2 ** 3$ results in 8 (2 raised to the power 3).

Lets take a look at an example script:

```
test.py x
test.py
1 # Demonstrating Arithmetic Operators in Python
2
3 # Addition (+)
4 print("Addition: 5 + 3 =", 5 + 3)
5
6 # Subtraction (-)
7 print("Subtraction: 10 - 4 =", 10 - 4)
8
9 # Multiplication (*)
10 print("Multiplication: 6 * 2 =", 6 * 2)
11
12 # Division (/) -> always gives a decimal (float)
13 print("Division: 8 / 2 =", 8 / 2)
14
15 # Floor Division (//) -> removes the decimal part
16 print("Floor Division: 8 // 3 =", 8 // 3)
17
18 # Modulus (%) -> gives the remainder
19 print("Modulus: 9 % 4 =", 9 % 4)
20
21 # Exponentiation (**) -> raises one number to the power of another
22 print("Exponentiation: 2 ** 3 =", 2 ** 3)
23
24 # -----
25 # Floats vs Ints
26 # -----
27 # An integer is a whole number like 5
28 # A float is a decimal number like 5.0
29
30 print("\nFloats vs Ints:")
31 print("5 is an integer:", 5)
32 print("5.0 is a float:", 5.0)
33 print("5 + 5.0 =", 5 + 5.0, "-> result is a float because one number was a float")
34

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER
7
(venv) doriannins@Dorians-MacBook-Studio TEST % /Users/doriannins/Desktop/TEST/venv/bin/python /Users/
File "/Users/doriannins/Desktop/TEST/test.py", line 2
result = 2 + 5 -
              ^
SyntaxError: invalid syntax
(venv) doriannins@Dorians-MacBook-Studio TEST % /Users/doriannins/Desktop/TEST/venv/bin/python /Users/
Addition: 5 + 3 = 8
Subtraction: 10 - 4 = 6
Multiplication: 6 * 2 = 12
Division: 8 / 2 = 4.0
Floor Division: 8 // 3 = 2
Modulus: 9 % 4 = 1
Exponentiation: 2 ** 3 = 8

Floats vs Ints:
5 is an integer: 5
5.0 is a float: 5.0
5 + 5.0 = 10.0 -> result is a float because one number was a float
(venv) doriannins@Dorians-MacBook-Studio TEST % []
```

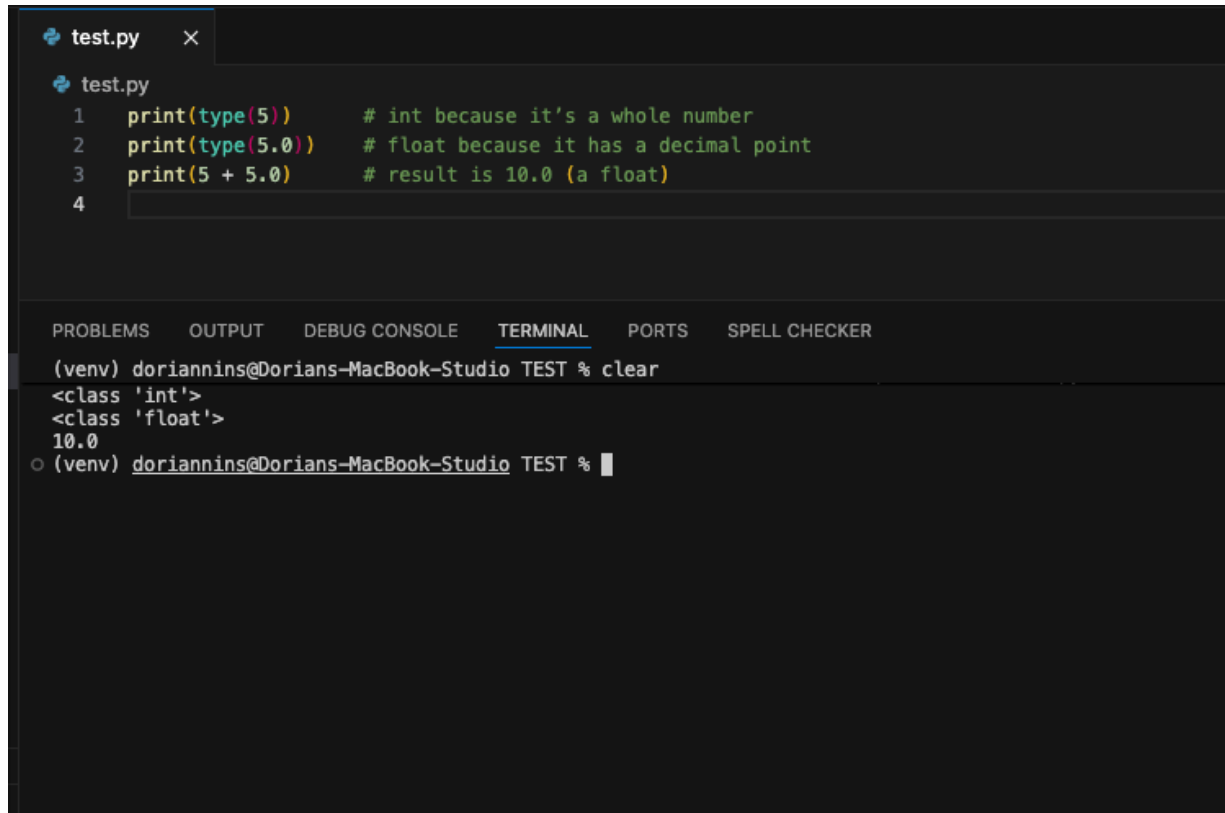
These operators work with both integers and floats, with Python automatically converting types as needed. For instance, dividing integers produces a float, but integer division truncates the decimal part.

FLOATS VS INTS?

In Python, there are two common kinds of numbers:

1. **Int** – short for “integer”
 - These are **whole numbers**.
 - Example: 1, 7, 0, -4

- They **don't have a dot**.
2. **Float** – short for “floating-point number”
- These are **numbers with a decimal point**.
 - Example: 1.5, 3.14, 5.0
 - Even if the decimal is .0, it's still a float.



The screenshot shows a code editor with a file named `test.py`. The code in the editor is as follows:

```
1 print(type(5))      # int because it's a whole number
2 print(type(5.0))    # float because it has a decimal point
3 print(5 + 5.0)      # result is 10.0 (a float)
4
```

Below the code editor, the **TERMINAL** tab is active, showing the output of running the script:

```
(venv) doriannins@Dorians-MacBook-Studio TEST % clear
<class 'int'>
<class 'float'>
10.0
(venv) doriannins@Dorians-MacBook-Studio TEST %
```

WHAT ARE VARIABLES

In Python, a **variable** is a name that you give to a value so you can store it and use it later. Think of it like a **container or a label** that holds something for you. Instead of writing the same number or value multiple times in your code, you can just use the variable.

Variables can store many types of data: numbers, text, lists, or even more complex things. For now, we'll focus on numbers. Variables let us **store a value once** and then **reuse it anywhere in our code**. This means we **don't have to write the same number or value multiple times**.

```
test.py ×
test.py > ...

4 # so we can use it later in calculations or display it.
5
6 # Example of variables
7 a = 10 # store 10 in a
8 b = 3  # store 3 in b
9
10 # Addition
11 sum_result = a + b
12 print("Addition:", a, "+", b, "=", sum_result)
13
14 # Subtraction
15 sub_result = a - b
16 print("Subtraction:", a, "-", b, "=", sub_result)
17
18 # Multiplication
19 mul_result = a * b
20 print("Multiplication:", a, "*", b, "=", mul_result)
21
22 # Division
23 div_result = a / b
24 print("Division:", a, "/", b, "=", div_result)
25
26 # Floor Division
27 floor_div_result = a // b
28 print("Floor Division:", a, "//", b, "=", floor_div_result)
29
30 # Modulus
31 mod_result = a % b
32 print("Modulus:", a, "%", b, "=", mod_result)
33
34 # Exponentiation
35 exp_result = a ** b
36 print("Exponentiation:", a, "**", b, "=", exp_result)
37

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER
(venv) doriannins@Dorians-MacBook-Studio TEST % clear
Addition: 10 + 3 = 13
Subtraction: 10 - 3 = 7
Multiplication: 10 * 3 = 30
Division: 10 / 3 = 3.3333333333333335
Floor Division: 10 // 3 = 3
Modulus: 10 % 3 = 1
Exponentiation: 10 ** 3 = 1000
(venv) doriannins@Dorians-MacBook-Studio TEST %
```

NUMERIC OPERATORS CHEAT

Operation Type	Python Symbol/ Method	Example Code	Output Explanation	Example Output
Addition	+	2 + 3	Adds two numbers	5

Subtraction	-	9 - 8	Subtracts second number from first	1
Multiplication	*	4 * 6	Multiplies two numbers	24
Division (Floating Point)	/	8 / 4	Divides and returns a float (decimal)	2.0
Division (Floating Point)	/	5 / 2	Divides, result includes decimal values	2.5
Integer Division	//	5 // 2	Divides and returns integer quotient only	2
Modulus (Remainder)	%	10 % 3	Returns remainder after division	1
Exponentiation (Power)	**	2 ** 3	Raises number to the power of another	8
Combined Operations	+, -, *, /	8 + 9 - 10	Combines multiple operations following PEMDAS	7
Operations with Parentheses	()	8 + (2 * 3)	Parentheses change order of operations	