Module 2 - Basic Python Concepts: Your First Steps

Welcome to the exciting world of Python programming! This course will guide you through the fundamental concepts that form the building blocks of almost any program you'll ever write. Get ready to think like a programmer and solve problems with code!

Chapter 1: Getting Started - The Console and Your First Program

This chapter will introduce you to the Python console (also known as the interactive interpreter) and guide you through writing and running your very first Python program: "Hello World!". We'll also look at common mistakes you might make.

1.1 The Python Console (Interactive Interpreter)

The **Python console** is an extremely useful tool for learning and testing small snippets of Python code directly. Think of it as a conversational partner for your Python code.

How to open the Python Console:

- **Windows:**
  - Search for "Python" in the Start menu. You might see "Python (Python 3.x)" or "IDLE (Python 3.x)".
  - **IDLE:** This is Python's Integrated Development and Learning Environment. It opens a console window where you can type Python commands.
  - **Command Prompt:** Open `cmd` or `PowerShell`, then type `python` and press Enter.
- **macOS:**
  - Open `Terminal` (Applications > Utilities > Terminal).
  - Type `python3` (or `python` on older systems) and press Enter.
- **Linux:**
  - Open `Terminal`.
  - Type `python3` (or `python`) and press Enter.

What you'll see:

Once you open the console, you'll typically see something like this:

```
Python 3.X.X (default, ...)
[GCC ...] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is the **prompt**. This indicates that the Python interpreter is ready to receive your commands.

Using the Console:

You can type Python code directly at the >>> prompt and press Enter. The interpreter will immediately execute the code and show you the result.

Example Console Interaction:

Python

```
>>> 2 + 2
4
>>> "Hello" + " " + "World"
'Hello World'
>>> 10 / 3
3.3333333333333335
>>> print("Learning Python is fun!")
Learning Python is fun!
>>>
```

Benefits of the Console:

- **Quick Testing:** Great for trying out new functions or syntax quickly.
- **Interactive Learning:** Provides immediate feedback, helping you understand how code works.
- **Debugging:** Can be used to inspect variable values during development.

1.2 Hello World! - Your First Program

Every journey begins with a first step, and in programming, that step is almost always printing "Hello World!". It's a simple tradition that helps you confirm your setup is working and gives you a taste of writing your first line of code.

What is it?

"Hello World!" is a classic introductory program that simply displays the text "Hello, World!" on your screen. This “Hello World” is framed by Brian Kernighan in C programming language in 1970.

Why is it important?

- **Verification:** It confirms that your Python interpreter is installed correctly, and you can run a basic script.
- **First Code:** It's your very first program – a significant milestone!
- **Output:** It introduces the concept of producing output from your program.

How to do it on the Console:

In Python, we use the `print()` function to display text.

Example Console Interaction:

Python
```
>>> print("Hello, World!")
Hello, World!
>>>
```

Explanation:

- `print()`: This is a built-in Python **function** that sends whatever is inside the parentheses to the standard output (usually your screen).
- `"Hello, World!"`: This is a "**string**" of text. Strings are sequences of characters enclosed in either single quotes (`'`) or double quotes (`"`).

How to do it in a File (Script):

While the console is great for quick tests, for larger programs, you'll write your code in a file (a script).

1. **Open a text editor:** You can use a simple one like Notepad (Windows), TextEdit (macOS - ensure it's plain text mode), or a code editor like VS Code or PyCharm (which we'll introduce later).
2. **Type the code:**

   Python

   ```
   # my_first_program.py
   print("Hello, World!")
   ```

3. **Save the file:** Save it with a `.py` extension (e.g., `hello_world.py`). Choose a memorable location like `Documents/PythonPrograms`.
4. **Run from Command Line/Terminal:**
   - Open your terminal or command prompt.
   - Navigate to the directory where you saved your file using the `cd` command (e.g., `cd Documents/PythonPrograms`).
   - Type `python hello_world.py` (or `python3 hello_world.py` on some systems) and press Enter.

Expected Output (for both console and file):

```
Hello, World!
```

1.3 Common Syntax Errors (and how to avoid them!)

Syntax errors are mistakes in the structure of your code that prevent Python from understanding and running it. They are like grammatical errors in a human language. Python will stop execution and tell you where it found the error.

Example 1: Missing Parentheses in `print()`

Python

```
>>> print "Hello World!"
  File "<stdin>", line 1
    print "Hello World!"
                       ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print("Hello World!")?
```

Explanation: In Python 3, `print` is a function and **requires parentheses** around its arguments. The error message is very helpful, even suggesting the correction!

Example 2: Mismatched Quotes

Python
```
>>> print('Hello, World!")
  File "<stdin>", line 1
    print('Hello, World!")
                          ^
SyntaxError: unterminated string literal (detected at line 1)
```

Explanation: You started the string with a single quote (`'`) but ended it with a double quote (`"`). Python expects the **same type of quote to close a string**.

Example 3: Typos in Keywords/Function Names

Python
```
>>> prit("Hello, World!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'prit' is not defined. Did you mean: 'print'?
```

Explanation: You misspelled `print`. Python doesn't know what `prit` means. Notice that this is a `NameError`, not a `SyntaxError`, because the syntax itself is valid, but the name `prit` doesn't exist. Python is often smart enough to suggest the correct name.

Example 4: Incorrect Indentation (Important in Python!)

Python uses **indentation** (spaces or tabs at the beginning of a line) to define code blocks. This is crucial! We'll explore it more when we cover control flow, but even basic errors can occur.

Python
```
>>>  print("Hello") # Notice the leading spaces here
  File "<stdin>", line 1
    print("Hello")
IndentationError: unexpected indent
```

Explanation: Python saw an indent where it didn't expect one. When you're just writing simple, top-level code, **do not indent** unless you're inside a function, loop, or conditional statement.

Key Takeaways for Syntax Errors:

- **Read the Error Message:** Python's error messages are often very informative. Look at the `ErrorType` (e.g., `SyntaxError`, `NameError`, `IndentationError`) and the message itself.
- **Look at the Line Number/Caret:** The error message usually tells you the file, line number, and even points with a `^` where it detected the problem.
- **Check for Typos:** Common culprits are misspelled function names or keywords.
- **Check for Mismatched Delimiters:** Parentheses `()`, square brackets `[]`, curly braces `{}`, and quotes `'' ""` must always come in matching pairs.
- **Pay Attention to Indentation:** Python is strict about it!

Chapter 2: Numbers and Mathematical Operations

Computers are excellent at crunching numbers! Python provides intuitive ways to perform basic mathematical operations.

2.1 Operators

These are the symbols we use to perform calculations.

Operator	Operation	Example	Console Output	Explanation
`+`	Addition	`5 + 3`	`8`	Standard addition.
`-`	Subtraction	`10 - 4`	`6`	Standard subtraction.
`*`	Multiplication	`7 * 2`	`14`	Standard multiplication.
`/`	Division	`15 / 3`	`5.0`	**Floating-point division** - always returns a `float` (decimal number), even if the result is a whole number. This is important for precision.
`//`	Floor Division	`17 // 3`	`5`	**Integer division** - divides and rounds *down* to the nearest whole number (integer). It effectively discards the fractional part.
`%`	Modulus	`17 % 3`	`2`	**Remainder of division**. This operator gives you the remainder after division. It's incredibly useful for checking if a number is even/odd (`num % 2 == 0`) or for cyclical operations (e.g., finding the hour on a 12-hour clock).
`**`	Exponentiation	`2 ** 3`	`8`	`base` raised to the power of `exponent` (e.g., `2 ** 3` means 2 multiplied by itself 3 times: 2 * 2 * 2).

Export to Sheets

Examples in the Console:

Python

```
>>> 10 + 5
15
>>> 20 - 7
13
>>> 4 * 6
24
>>> 100 / 4
25.0
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 17 % 5
2
>>> 3 ** 4 # 3 * 3 * 3 * 3
81
>>>
```

2.2 Order of Operations (PEMDAS/BODMAS)

Python follows the standard mathematical order of operations, just like in algebra:

1. **P**arentheses / **B**rackets: Operations inside parentheses are performed first.
2. **E**xponents / **O**rders: Powers are calculated next.
3. **M**ultiplication and **D**ivision: These are performed from left to right.
4. **A**ddition and **S**ubtraction: These are performed last, from left to right.

Example in the Console:

Python
```
>>> 5 + 2 * 3
11             # Explanation: Multiplication (2 * 3 = 6) happens before
addition (5 + 6 = 11)

>>> (5 + 2) * 3
21             # Explanation: Parentheses are evaluated first (5 + 2 = 7),
then multiplication (7 * 3 = 21)

>>> 10 - 4 / 2
8.0            # Explanation: Division (4 / 2 = 2.0) happens before
subtraction (10 - 2.0 = 8.0)

>>> 2 ** 3 + 1
9              # Explanation: Exponentiation (2 ** 3 = 8) happens before
addition (8 + 1 = 9)

>>> 20 / 4 * 2 # Explanation: Division and multiplication have the same
precedence, so left to right
10.0           # (20 / 4 = 5.0), then (5.0 * 2 = 10.0)
>>>
```

2.3 Common Numeric Syntax Errors

- **Division by Zero:** This is a mathematical impossibility and will cause an error known as `ZeroDivisionError`.

Python

```
>>> 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Explanation: Python explicitly tells you it's a `ZeroDivisionError`. Always ensure your denominator is not zero.

- **Missing Operators:** You can't just put numbers next to each other to imply multiplication.

  Python

  ```
  >>> 5(2+3)
  SyntaxError: cannot call non-function of type 'int'
  ```

  Explanation: Python thinks `5` is a function you're trying to call with `(2+3)` as arguments. You need an explicit multiplication operator `*`. **Correct:** `5 * (2 + 3)`

Chapter 3: Strings - Working with Text

Strings are fundamental for working with text data in Python. They are sequences of characters.

3.1 Defining Strings

You can define strings using **single quotes (`'`)** or **double quotes (`"`)**. It's generally good practice to be consistent within your code.

Examples in the Console:

Python
```
>>> 'Hello, Python!'
'Hello, Python!'
>>> "I love programming."
'I love programming.'
>>> type("Hello") # Check the data type
<class 'str'>
>>>
```

Why have both?

If your string needs to contain a quote of the *other* type, you don't need to escape it. This makes your code cleaner.

Python
```
>>> message1 = "He said, 'Hello!'"
>>> print(message1)
```

```
He said, 'Hello!'
>>> message2 = 'She replied, "Hi there!"'
>>> print(message2)
She replied, "Hi there!"
>>>
```

3.2 Multiline Strings

For strings that span multiple lines, use **triple quotes** (either `'''` or `"""`). The newline characters are preserved.

Example:

Python
```
>>> multiline_string = """This is a string
... that spans
... multiple lines."""
>>> print(multiline_string)
This is a string
that spans
multiple lines.
>>>
```

(Note: In the console, when you press Enter after the first line of a triple-quoted string, you'll see `...` which indicates Python is expecting more input for the same string until you close the triple quotes.)

3.3 Escaping Characters

Sometimes you need to include special characters in your string, like a newline or a tab, or a quote of the *same* type that you used to define the string. We use a **backslash (\)** followed by a character to "**escape**" it, giving it a special meaning.

Escape Sequence	Meaning	Console Example	Output
`\n`	Newline	`print("Line 1\nLine 2")`	`Line 1&lt;br>Line 2`
`\t`	Tab	`print("Name:\tAlice")`	`Name: Alice`
`\\`	Backslash	`print("Path: C:\\Users\\Name")`	`Path: C:\Users\Name`
`\'`	Single Quote	`print('It\'s a sunny day.')`	`It's a sunny day.`
`\"`	Double Quote	`print("He said, \"Hello!\"")`	`He said, "Hello!"`

Export to Sheets

Examples in the Console:

Python
```
>>> print("First line.\nSecond line.")
First line.
Second line.
>>> print("Item\tPrice")
Item    Price
```

```
>>> print("This is a backslash: \\")
This is a backslash: \
>>> print('I\'m learning Python.')
I'm learning Python.
>>>
```

3.4 Common String Syntax Errors

- **Unterminated String Literal:** Forgetting to close a string with the matching quote.

  Python

  ```
  >>> message = "Hello, World!
  SyntaxError: unterminated string literal (detected at line 1)
  ```

  Explanation: You opened the string with `"` but never closed it. Python reaches the end of the line (or file) and finds an open string.

- **Using backslashes incorrectly (especially on Windows paths):** When dealing with file paths, especially on Windows, backslashes have special meanings. If you're not careful, Python might interpret them as escape sequences.

  Python

  ```
  >>> path = "C:\Users\Name\Desktop"
  SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes
  in position 2-3: truncated \UXXXXXXXX escape
  ```

  Explanation: `\U` is an escape sequence for Unicode characters. When Python sees `\U` it expects a certain number of following characters to form a valid Unicode code point. Since `\U` is incomplete, it's a syntax error. **Solution:** Use **raw strings** (prefix with `r`) or **double backslashes `\\`**.

  Python

  ```
  >>> path = r"C:\Users\Name\Desktop" # Raw string - treats backslashes
  literally
  >>> print(path)
  C:\Users\Name\Desktop

  >>> path = "C:\\Users\\Name\\Desktop" # Double backslashes - escapes
  the backslash itself
  >>> print(path)
  C:\Users\Name\Desktop
  >>>
  ```

Chapter 4: Input Function - Getting User Input

Programs aren't just about output; they often need to interact with the user! The `input()` function allows your program to pause and wait for the user to type something and press Enter.

4.1 How it Works

The `input()` function takes an optional argument: a **prompt string** that is displayed to the user. Whatever the user types is captured and returned as a **string**.

Example in a Script:

Let's create a file named `get_name.py` and run it from your terminal/command prompt:

Python
```
# get_name.py
print("Welcome to our program!")
name = input("What is your name? ")
print("Hello, " + name + "!")

age_str = input("How old are you? ")
print("You are " + age_str + " years old.")
```

Running `get_name.py` from the Terminal:

```
python get_name.py
```

Example Interaction in the Console/Terminal:

```
Welcome to our program!
What is your name? Alice
Hello, Alice!
How old are you? 30
You are 30 years old.
```

Important Note: `input()` always returns a string!

This is a crucial point for beginners. Even if the user types numbers, `input()` treats them as text.

Python
```
>>> num_str = input("Enter a number: ")
Enter a number: 123
>>> print(type(num_str))
<class 'str'>
>>>
```

4.2 Type Conversion (Casting)

If you want to perform mathematical operations on the input, you need to convert it to a numeric type (like `int` for integers or `float` for decimal numbers) using **type casting functions**.

- `int()`: Converts a string (or float) to an integer.
- `float()`: Converts a string (or integer) to a floating-point number.

Example in a Script:

Let's create a file named `simple_calculator.py`:

Python
```
# simple_calculator.py
print("--- Simple Calculator ---")

num1_str = input("Enter the first number: ")
num2_str = input("Enter the second number: ")

# Convert strings to integers using int()
num1 = int(num1_str)
num2 = int(num2_str)

sum_of_numbers = num1 + num2
print("The sum of", num1, "and", num2, "is:", sum_of_numbers)

# Example with float()
price_str = input("Enter an item's price (e.g., 12.99): ")
price = float(price_str)
print("The price you entered is: $", price)
```

Example Interaction (running `simple_calculator.py`):

```
--- Simple Calculator ---
Enter the first number: 10
Enter the second number: 25
The sum of 10 and 25 is: 35
Enter an item's price (e.g., 12.99): 45.75
The price you entered is: $ 45.75
```

4.3 Common Input-Related Errors

- **`ValueError` when converting non-numeric input:** If the user types something that *cannot* be converted to an `int` or `float` (e.g., "hello" when expecting a number), your program will crash with a `ValueError`.

  Python

  ```
  >>> age_str = input("Enter your age: ")
  Enter your age: twenty
  >>> age = int(age_str)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: invalid literal for int() with base 10: 'twenty'
  ```

  Explanation: Python tried to convert the string `'twenty'` into an integer but couldn't because it's not a valid integer representation. This is a common runtime error for beginner programs that rely on user input. Later, you'll learn about `try-except` blocks to handle such errors gracefully without crashing your program.

Chapter 5: String Operations - Manipulating Text

Strings are incredibly versatile! Python provides many ways to manipulate and combine them.

5.1 Concatenation (Joining Strings)

Use the `+` operator to join two or more strings together. This is called **concatenation**.

Example in the Console:

Python
```
>>> "Hello" + "World"
'HelloWorld'
>>> "Hello" + " " + "World" # Adding a space for readability
'Hello World'
>>> first_name = "Jane"
>>> last_name = "Doe"
>>> full_name = first_name + " " + last_name
>>> print(full_name)
Jane Doe
>>>
```

5.2 Repetition

Use the `*` operator with an integer to repeat a string multiple times.

Example in the Console:

Python
```
>>> "Python" * 3
'PythonPythonPython'
>>> border = "=" * 20
>>> print(border)
====================
>>>
```

5.3 String Length

The `len()` function returns the number of characters in a string.

Example in the Console:

Python
```
>>> my_string = "Programming"
>>> len(my_string)
11
>>> len("Hello")
5
>>> len("") # Length of an empty string
0
>>>
```

5.4 Indexing (Accessing Characters)

Each character in a string has an **index**, starting from `0` for the first character.

Example in the Console:

Python
```
>>> word = "PYTHON"
>>> word[0] # First character (index 0)
'P'
>>> word[1] # Second character (index 1)
'Y'
>>> word[5] # Sixth character (index 5) - the last character
'N'

>>> # Negative indexing (from the end)
>>> word[-1] # Last character
'N'
>>> word[-2] # Second to last character
'O'
>>>
```

Important: Trying to access an index that doesn't exist (e.g., trying to get the 10th character of a 5-character string) will result in an `IndexError`.

Python
```
>>> word = "ABC"
>>> word[3] # Index 3 is out of bounds for a 3-character string (valid
indices are 0, 1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

5.5 Slicing (Extracting Substrings)

You can extract a portion (a "slice") of a string using the syntax: `[start:end:step]`.

- `start`: The index where the slice begins (**inclusive**). If omitted, defaults to 0 (the beginning).
- `end`: The index where the slice ends (**exclusive** - the character at this index is NOT included). If omitted, defaults to the end of the string.
- `step`: The step size (e.g., `2` for every second character). If omitted, defaults to 1.

Examples in the Console:

Python
```
>>> sentence = "Hello, Python Programming!"

>>> sentence[0:5]    # Characters from index 0 up to (but not including) 5
'Hello'
>>> sentence[7:13]   # Characters from index 7 up to (but not including) 13
'Python'
>>> sentence[:5]     # From beginning to index 5 (exclusive)
'Hello'
>>> sentence[7:]     # From index 7 to the end
'Python Programming!'
>>> sentence[::2]    # Every second character
'Hlo yhnPormig!'
>>> sentence[::-1]   # Reverse the string (step of -1 means go backwards)
'!gnimmargorP nohtyP ,olleH'
```

```
>>>
```

5.6 String Methods

Strings have many useful built-in **methods** (functions that "belong to" the string object and perform operations on it). You call them using the **dot (`.`) operator**: `string_variable.method_name()`.

- `.upper()`: Converts all characters in the string to uppercase.
- `.lower()`: Converts all characters in the string to lowercase.
- `.capitalize()`: Capitalizes the first character of the string, and converts the rest to lowercase.
- `.title()`: Capitalizes the first letter of each word in the string.
- `.strip()`: Removes leading and trailing whitespace (spaces, tabs, newlines) from the string.
- `.replace(old, new)`: Returns a *new* string with all occurrences of `old` substring replaced by `new` substring.
- `.find(substring)`: Returns the lowest index in the string where the `substring` is found. Returns `-1` if the substring is not found.
- `.count(substring)`: Returns the number of non-overlapping occurrences of `substring` in the string.

Examples in the Console:

Python
```
>>> text = "   Hello, World!   "

>>> text.upper()
'   HELLO, WORLD!   '
>>> text.lower()
'   hello, world!   '
>>> text.strip()
'Hello, World!'
>>> text.replace("World", "Python")
'   Hello, Python!   '

>>> "hello world".capitalize()
'Hello world'
>>> "hello world".title()
'Hello World'

>>> "apple".find("p")
1
>>> "banana".count("a")
3
>>> "hello".find("xyz") # Substring not found
-1
>>>
```

5.7 Common String Operation Errors

- **`TypeError` when mixing string and non-string types with `+`:**

  Python

```
>>> "My age is " + 30
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Explanation: You cannot directly concatenate a string and an integer using `+`. You must convert the number to a string first using `str()`. **Correct:** `"My age is " + str(30)`

- **`TypeError` when using `*` with non-integer:**

  Code snippet

```
>>> "Hello" * "World"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

  Explanation: The repetition operator `*` only works with a string and an integer (specifying how many times to repeat).

- **Strings are Immutable:** You cannot change individual characters in a string using indexing.

  Python

```
>>> my_string = "hello"
>>> my_string[0] = "H"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

  Explanation: Strings are "**immutable**" meaning their contents cannot be changed *after they are created*. If you need a modified string, you create a *new* string based on the old one. **Correct:** `my_string = "H" + my_string[1:]` (This creates a new string by concatenating "H" with a slice of the rest of the original string).

Chapter 6: Variables - Storing Data

Variables are like named containers or labels that hold values. They are essential for storing and manipulating data in your programs.

6.1 Why use Variables?

- **Storage:** To store data (numbers, text, True/False values, etc.) that your program will use.
- **Readability:** To make your code more understandable by giving meaningful names to data (e.g., `user_name` instead of just `"Alice"`). This is crucial for larger programs.
- **Flexibility:** To easily change values in one place, affecting multiple parts of your code. If a value (like a tax rate or a user's age) changes, you only need to update it

once where it's stored in the variable, rather than searching for every instance of that value.

6.2 Creating Variables

You create a variable by giving it a name and assigning a value to it using the **assignment operator (=)**.

Examples in the Console:

Python
```
>>> age = 30            # 'age' is an integer variable
>>> name = "Alice"      # 'name' is a string variable
>>> price = 19.99       # 'price' is a float variable
>>> is_student = True   # 'is_student' is a boolean variable (True/False are special keywords)

>>> print(age)
30
>>> print(name)
Alice
>>> print(price)
19.99
>>> print(is_student)
True

>>> type(age)        # Check the data type of the variable
<class 'int'>
>>> type(name)
<class 'str'>
>>>
```

6.3 Variable Naming Rules (Important!)

Python has specific rules for naming variables. If you break these rules, you'll get a `SyntaxError` or `NameError`.

1. **Start with a letter or underscore (_):**
   - `my_variable` (Valid)
   - `_temp` (Valid)
   - `1variable` (Invalid - `SyntaxError` because it starts with a number)
2. **Contain only letters, numbers, and underscores:**
   - `user_id` (Valid)
   - `data_2` (Valid)
   - `my-variable` (Invalid - hyphen is treated as subtraction, leading to `SyntaxError`)
3. **Case-sensitive:** `age` is different from `Age` and `AGE`.

   Python

   ```
   >>> myVar = 10
   >>> print(myvar)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   ```

```
NameError: name 'myvar' is not defined. Did you mean: 'myVar'?
```

Explanation: Python treats `myVar` and `myvar` as two completely separate variables. Be consistent with your casing!

4. **Cannot be Python keywords:** You cannot use words that have special meaning in Python (like `if`, `for`, `while`, `print`, `True`, `False`, `None`, etc.).

   Python

   ```
   >>> for = 10
   SyntaxError: invalid syntax
   ```

   Common Keywords (don't use these as variable names): `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`

6.4 Good Naming Conventions (Recommended)

While not strict rules, these are widely adopted practices that make your code readable and maintainable, especially when working with others.

- **Snake Case (most common in Python):** Words are lowercase and separated by underscores.
  - `first_name`
  - `total_score`
  - `is_active_user`
- **Descriptive names:** Choose names that clearly indicate what the variable holds. Avoid single-letter variable names unless their purpose is absolutely obvious (e.g., `x`, `y` for coordinates in a mathematical context).
  - **Good:** `customer_address`, `product_price_usd`
  - **Bad:** `ca`, `pp`

6.5 Reassigning Variables

You can change the value of a variable at any time. The variable will then hold the new value, overwriting the old one.

Python

```
>>> score = 100
>>> print("Initial score:", score)
Initial score: 100

>>> score = 150 # Reassigned to a new integer value
>>> print("New score:", score)
New score: 150

>>> message = "Hello"
>>> message = "Goodbye" # Can even change the type of value stored (though
often not best practice in simple scripts)
>>> print(message)
Goodbye
```

```
>>>
```

6.6 Multiple Assignment

You can assign multiple variables on a single line, making your code more concise.

Python
```
>>> x, y, z = 10, 20, 30
>>> print(x, y, z)
10 20 30

>>> a = b = c = "Hello" # All three variables now refer to the same string
object
>>> print(a, b, c)
Hello Hello Hello
>>>
```

6.7 Common Variable Errors

- **`NameError: name '...' is not defined`**: This happens when you try to use a variable before you've assigned a value to it, or if you've misspelled the variable name. This is a very common error for beginners!

  Python

  ```
  >>> print(my_variable)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'my_variable' is not defined
  ```

  Explanation: Python doesn't know what `my_variable` refers to because it hasn't been created (assigned a value) yet. Always assign a value to a variable before you try to use it.

Chapter 7: Increment/Decrement - Changing Variable Values

Incrementing means increasing a variable's value, and **decrementing** means decreasing it. While Python doesn't have the `++` or `--` operators found in some other languages (like C++ or Java), it offers clear and concise ways to achieve this.

7.1 Standard Way

To increment a variable, you simply add a value to it and reassign the result to the same variable.

Examples in the Console:

Python

```
>>> counter = 0
>>> print("Initial counter:", counter)
Initial counter: 0

>>> # Increment by 1
>>> counter = counter + 1
>>> print("Counter after incrementing by 1:", counter)
Counter after incrementing by 1: 1

>>> # Increment by 5
>>> counter = counter + 5
>>> print("Counter after incrementing by 5:", counter)
Counter after incrementing by 5: 6
>>>
```

Similarly for decrementing:

Python

```
>>> lives = 3
>>> print("Initial lives:", lives)
Initial lives: 3

>>> # Decrement by 1
>>> lives = lives - 1
>>> print("Lives after decrementing by 1:", lives)
Lives after decrementing by 1: 2

>>> # Decrement by 0.5
>>> lives = lives - 0.5
>>> print("Lives after decrementing by 0.5:", lives)
Lives after decrementing by 0.5: 1.5
>>>
```

7.2 Shorthand Assignment Operators (Augmented Assignment)

Python provides convenient shorthand operators for common operations like addition, subtraction, multiplication, and division combined with assignment. These are very frequently used because they are more concise and often more efficient!

Operator	Equivalent To	Example in Console	Result	Explanation
`+=`	`x = x + y`	`x = 10; x += 5`	`x` is `15`	Add `y` to `x`, then assign the new value to `x`.
`-=`	`x = x - y`	`x = 10; x -= 2`	`x` is `8`	Subtract `y` from `x`, then assign the new value to `x`.
`*=`	`x = x * y`	`x = 10; x *= 3`	`x` is `30`	Multiply `x` by `y`, then assign the new value to `x`.
`/=`	`x = x / y`	`x = 10; x /= 4`	`x` is `2.5`	Divide `x` by `y`, then assign the new value to `x`. Returns float.
`//=`	`x = x // y`	`x = 10; x //= 4`	`x` is `2`	Floor divide `x` by `y`, then assign the new value to `x`.
`%=`	`x = x % y`	`x = 10; x %= 3`	`x` is `1`	Get remainder of `x` divided by `y`, then assign the new value to `x`.

`**=`	`x = x ** y`	`x = 2; x **= 3`	`x` is `8`	Raise `x` to the power of `y`, then assign the new value to `x`.

Export to Sheets

More Examples in the Console:

Python

```
>>> points = 100
>>> points += 10 # Add 10 to points
>>> print("Points after += 10:", points)
Points after += 10: 110

>>> energy = 50
>>> energy -= 5  # Subtract 5 from energy
>>> print("Energy after -= 5:", energy)
Energy after -= 5: 45

>>> price = 25
>>> price *= 1.1 # Increase price by 10% (multiply by 1.1)
>>> print("Price after *= 1.1:", price)
Price after *= 1.1: 27.5

>>> count = 7
>>> count //= 2 # Integer division of count by 2
>>> print("Count after //= 2:", count)
Count after //= 2: 3

>>> score = 100
>>> bonus_multiplier = 1.5
>>> score *= bonus_multiplier # Using a variable for the amount to change
by
>>> print("Score after *= bonus_multiplier:", score)
Score after *= bonus_multiplier: 150.0
>>>
```

7.3 Common Increment/Decrement Errors

- **Confusing + with += (for beginners):** A common mistake is to write `x + 1` thinking it will update the value of `x`. It does not! `x + 1` is an *expression* that produces a result, but that result is not stored back into `x` unless you explicitly assign it.

  Python

  ```
  >>> x = 5
  >>> x + 1 # This calculates x + 1 (which is 6) but doesn't change x
  itself
  6
  >>> print(x) # x is still 5
  5
  ```

  Explanation: `x + 1` evaluates to `6`, but its result is not stored anywhere. To update `x`, you *must* use an assignment: `x = x + 1` or, more commonly, `x += 1`.

Chapter 8: Intro to PyCharm - Your Development Environment

While you can write and run Python code using a simple text editor and the command line, an **Integrated Development Environment (IDE)** like PyCharm significantly enhances your coding experience.

8.1 What is an IDE?

An IDE is a software application that provides comprehensive facilities to computer programmers for software development. It typically consists of:

- **Source code editor:** Where you write your code with helpful features like syntax highlighting and code completion.
- **Build automation tools:** For interpreting or compiling your code and managing project dependencies.
- **Debugger:** For finding and fixing errors in your code systematically.
- **Other features:** Project management, version control integration, etc.

8.2 Why PyCharm?

PyCharm is a popular and powerful IDE specifically designed for Python. It offers a wide range of features that make coding easier, faster, and more enjoyable:

- **Intelligent Code Editor:**
    - **Syntax Highlighting:** Automatically colors different parts of your code (keywords, strings, comments) for better readability and quick identification of elements.
    - **Code Completion (IntelliSense):** Suggests function names, variable names, and parameters as you type, significantly speeding up coding and reducing typos.
    - **Error Highlighting:** Catches syntax errors and potential logical issues *before* you even run your code, underlining them in red or yellow, helping you fix problems early.
    - **Code Formatting:** Automatically formats your code to adhere to Python's widely accepted style guidelines (PEP 8), ensuring consistent and readable code.
    - **Refactoring Tools:** Helps you safely rename variables, functions, or move code blocks across your entire project without breaking references.
- **Integrated Debugger:** This is an incredibly powerful tool for understanding and fixing problems in your code. You can:
    - **Set breakpoints:** Pause your program's execution at specific lines.
    - **Step through code:** Execute your code line by line.
    - **Inspect variables:** View the values of all your variables at any point during execution.
    - This systematic approach makes finding and fixing complex bugs much easier than just using `print()` statements.

- **Project Management:** PyCharm organizes your files and folders into logical projects, keeping your code structured and easy to navigate, especially for larger applications.
- **Version Control Integration:** Seamlessly works with popular version control systems like Git. This allows you to track changes to your code over time, revert to previous versions, and collaborate effectively with other developers.
- **Virtual Environments:** PyCharm makes it easy to set up and manage **virtual environments** for your projects. This isolates project-specific libraries and dependencies, preventing conflicts between different projects on your machine. For example, Project A might need `library_x` version 1, while Project B needs `library_x` version 2 – virtual environments handle this smoothly.

8.3 Getting Started with PyCharm (Community Edition is Free!)

The **PyCharm Community Edition** is free and open-source, making it perfect for learning and small projects.

1. **Download and Install:**
    - Go to the official JetBrains PyCharm website: https://www.jetbrains.com/pycharm/
    - Click "Download" and select the **Community Edition**.
    - Follow the installation instructions for your operating system (Windows, macOS, or Linux).
2. **Create a New Project:**
    - Open PyCharm.
    - On the Welcome screen, select "**New Project**".
    - **Location:** Choose a folder on your computer where you want to store your Python projects (e.g., `C:\Users\YourName\PyCharmProjects\`). Give your project a meaningful name (e.g., `MyFirstPythonProject`).
    - **Python Interpreter:** PyCharm will usually suggest creating a new "**Virtual environment**". This is a good default practice. It will create a fresh, isolated Python installation for your project. Ensure the "Base interpreter" points to your main Python installation (e.g., `Python 3.10` or whatever version you installed).
    - Click "**Create**".
3. **Create a New Python File:**
    - In the PyCharm project view (the left-hand panel, usually showing your project name), right-click on your project name.
    - Go to `New` > `Python File`.
    - Give your file a descriptive name (e.g., `my_script.py`, `calculator.py`). Press Enter.
    - A new, empty `.py` file will open in the main editor area.
4. **Write and Run Code:**
    - Type your Python code directly in the editor window (e.g., `print("Hello from PyCharm!")`).
    - To run your code:
        - **Right-click anywhere in the editor window** and select `Run 'your_file_name'` from the context menu.
        - Or, click the **green "Play" button** (often a triangle icon) in the top right corner of the PyCharm window.

    - The output of your program will appear in the "**Run**" window at the bottom of the PyCharm interface.

Key Benefits of PyCharm for Students:

- **Faster Learning:** The intelligent features (code completion, error highlighting) reduce frustration from syntax errors and help you learn syntax quicker and more accurately.
- **Professional Workflow:** Gets you accustomed to tools and practices used in professional software development, preparing you for future roles.
- **Debugging Skills:** Learning to use a debugger effectively is an essential skill for any programmer, and PyCharm's debugger is intuitive and powerful.

Chapter 9: Coding Challenges - Applied Practice

Now it's your turn to apply what you've learned! These challenges are based on common practical scenarios and will help solidify your understanding. Try to solve them first on your own, and then you can compare your solutions or ask for help if you get stuck!

Challenge 1: Temperature Converter

Goal: Create a program that converts temperatures between Celsius and Fahrenheit.

Concepts Covered:

- `input()` for user interaction.
- Type conversion (`float()`) for calculations.
- Arithmetic operators (`*`, `/`, `+`, `-`).
- `print()` for displaying results.

Requirements:

1. **Celsius to Fahrenheit:**
    - Prompt the user to enter a temperature in Celsius.
    - Convert the input string to a `float`.
    - Apply the formula: `Fahrenheit = (Celsius * 9/5) + 32`
    - Print the result clearly.
2. **Fahrenheit to Celsius:**
    - Prompt the user to enter a temperature in Fahrenheit.
    - Convert the input string to a `float`.
    - Apply the formula: `Celsius = (Fahrenheit - 32) * 5/9`
    - Print the result clearly.

Example Script (`temperature_converter.py`):

Python
```
# --- Celsius to Fahrenheit Conversion ---
```

```
print("\n--- Celsius to Fahrenheit Converter ---")
celsius_str = input("Enter temperature in Celsius: ")
celsius = float(celsius_str)
fahrenheit = (celsius * 9/5) + 32
print(f"Fahrenheit: {fahrenheit}°F") # Using an f-string for cleaner output

# --- Fahrenheit to Celsius Conversion ---
print("\n--- Fahrenheit to Celsius Converter ---")
fahrenheit_str = input("Enter temperature in Fahrenheit: ")
fahrenheit = float(fahrenheit_str)
celsius = (fahrenheit - 32) * 5/9
print(f"Celsius: {celsius}°C") # Using an f-string for cleaner output
```

Example Interaction:

```
--- Celsius to Fahrenheit Converter ---
Enter temperature in Celsius: 25
Fahrenheit: 77.0°F

--- Fahrenheit to Celsius Converter ---
Enter temperature in Fahrenheit: 68
Celsius: 20.0°C
```

Note on `f-strings` (Formatted String Literals): You'll notice `f"Fahrenheit: {fahrenheit}°F"`. This is an **f-string**, a very convenient way to embed expressions inside string literals. You prefix the string with `f` (or `F`) and then place variable names or expressions inside curly braces `{}` within the string. Python will automatically replace them with their values. It's generally preferred over string concatenation (`+`) for readability.

Challenge 2: Simple Interest Calculator

Goal: Calculate the Simple Interest (SI) given Principal, Number of years, and Rate of interest.

Concepts Covered:

- `input()` for collecting multiple pieces of data.
- Type conversion (`int()` or `float()`) for numeric input.
- Arithmetic operators (`*`, `/`).
- `print()` for displaying the final result.

Requirements:

1. Print a clear heading for the "Simple Interest Calculator".
2. Define the formula: `SI = (P * N * R) / 100`
    - `P`: Principal amount
    - `N`: Number of years
    - `R`: Rate of interest
3. Prompt the user to enter the **Principal (P)**.
4. Prompt the user to enter the **Number of years (N)**.
5. Prompt the user to enter the **Rate of interest (R)**.

6. **Convert all inputs** to appropriate numeric types (`float()` is often safer here as rates/principals can be decimal).
7. Calculate the Simple Interest using the formula.
8. Print the calculated Simple Interest.

Example Script (`si_calculator.py`):

Python
```
# --- SI CALCULATOR ---
print("\n--- Simple Interest Calculator ---")
print("Formula: SI = (P * N * R) / 100")

# Get inputs as strings first
principal_str = input("Enter Principal (P): ")
years_str = input("Enter Number of years (N): ")
rate_str = input("Enter Rate of interest (R): ")

# Convert inputs to float for calculations
principal = float(principal_str)
years = float(years_str)
rate = float(rate_str)

# Calculate Simple Interest
simple_interest = (principal * years * rate) / 100

# Print the result
print(f"The Simple Interest (SI) is: {simple_interest}")
```

Example Interaction:

```
--- Simple Interest Calculator ---
Formula: SI = (P * N * R) / 100
Enter Principal (P): 1000
Enter Number of years (N): 2
Enter Rate of interest (R): 5
The Simple Interest (SI) is: 100.0
```

Chapter 10: Numeric Functions - Advanced Math Operations

Besides the basic arithmetic operators, Python offers several built-in functions to perform common numeric operations. These are very useful for a wide range of calculations.

10.1 `abs()` (Absolute Value)

Returns the absolute value of a number (its distance from zero, always positive).

Examples in the Console:

Python
```
>>> abs(-10)
10
>>> abs(5.7)
```

```
5.7
>>> abs(0)
0
>>>
```

10.2 `round()` (Rounding)

Rounds a number to the nearest integer or to a specified number of decimal places.

- `round(number)`: Rounds to the nearest integer.
- `round(number, num_digits)`: Rounds to `num_digits` after the decimal point.

Examples in the Console:

Python
```
>>> round(3.14159)
3
>>> round(3.7)
4
>>> round(2.5) # Python's round() for .5 rounds to the nearest even number
2
>>> round(3.5)
4
>>> round(3.14159, 2) # Round to 2 decimal places
3.14
>>> round(123.456, 1) # Round to 1 decimal place
123.5
>>>
```

10.3 `min()` (Minimum Value)

Returns the smallest item in an iterable (like a list or tuple) or the smallest of two or more arguments.

Examples in the Console:

Python
```
>>> min(10, 5, 20, 3) # With multiple arguments
3
>>> numbers = [10, 5, 20, 3]
>>> min(numbers) # With a list
3
>>> min(-1, -5, 0)
-5
>>>
```

10.4 `max()` (Maximum Value)

Returns the largest item in an iterable or the largest of two or more arguments.

Examples in the Console:

Python
```
>>> max(10, 5, 20, 3) # With multiple arguments
```

```
20
>>> numbers = [10, 5, 20, 3]
>>> max(numbers) # With a list
20
>>> max(100, 1000, 50)
1000
>>>
```

10.5 `sum()` (Sum of Items)

Returns the sum of all items in an iterable (like a list or tuple).

Examples in the Console:

Python
```
>>> grades = [85, 90, 78, 92, 88]
>>> sum(grades)
433
>>> print(sum([1, 2, 3]))
6
>>>
```

10.6 `pow(base, exp)` (Power Function)

Equivalent to `base ** exp`. Can also take an optional third argument `mod` (which calculates `(base ** exp) % mod`).

Examples in the Console:

Python
```
>>> pow(2, 3)   # 2 to the power of 3
8
>>> pow(5, 2)   # 5 squared
25
>>> pow(4, 0.5) # Square root (4 to the power of 0.5)
2.0
>>>
```

10.7 `divmod(numerator, denominator)` (Quotient and Remainder)

Returns a tuple containing the quotient and remainder of the division. This is often more efficient than calculating `//` and `%` separately if you need both.

Examples in the Console:

Python
```
>>> divmod(17, 3) # Returns (quotient, remainder)
(5, 2)
>>> quotient, remainder = divmod(25, 4)
>>> print("Quotient:", quotient)
Quotient: 6
>>> print("Remainder:", remainder)
Remainder: 1
>>>
```

10.8 Practical Application Example

Let's use some of these functions to calculate an average and find extreme values.

Example in a Script:

Python
```
# calculate_stats.py
print("--- Grade Statistics ---")

# Get grades as input
grade1 = float(input("Enter grade 1: "))
grade2 = float(input("Enter grade 2: "))
grade3 = float(input("Enter grade 3: "))

# Put grades into a list for sum/min/max functions
grades = [grade1, grade2, grade3]

# Calculate sum and average
total_grades = sum(grades)
num_grades = len(grades) # len() works for strings and lists!
average_grade = total_grades / num_grades

# Find highest and lowest
highest_grade = max(grades)
lowest_grade = min(grades)

# Round the average for display
rounded_average = round(average_grade, 2)

print("\n--- Results ---")
print("Entered grades:", grades)
print("Total points:", total_grades)
print("Average grade:", rounded_average)
print("Highest grade:", highest_grade)
print("Lowest grade:", lowest_grade)
```

Example Interaction (running `calculate_stats.py`):

```
--- Grade Statistics ---
Enter grade 1: 85.5
Enter grade 2: 92
Enter grade 3: 78.3

--- Results ---
Entered grades: [85.5, 92.0, 78.3]
Total points: 255.8
Average grade: 85.27
Highest grade: 92.0
Lowest grade: 78.3
```

Remember to practice regularly! The more you code, the better you'll become. Experiment with these concepts in the console and by writing small scripts. Good luck!