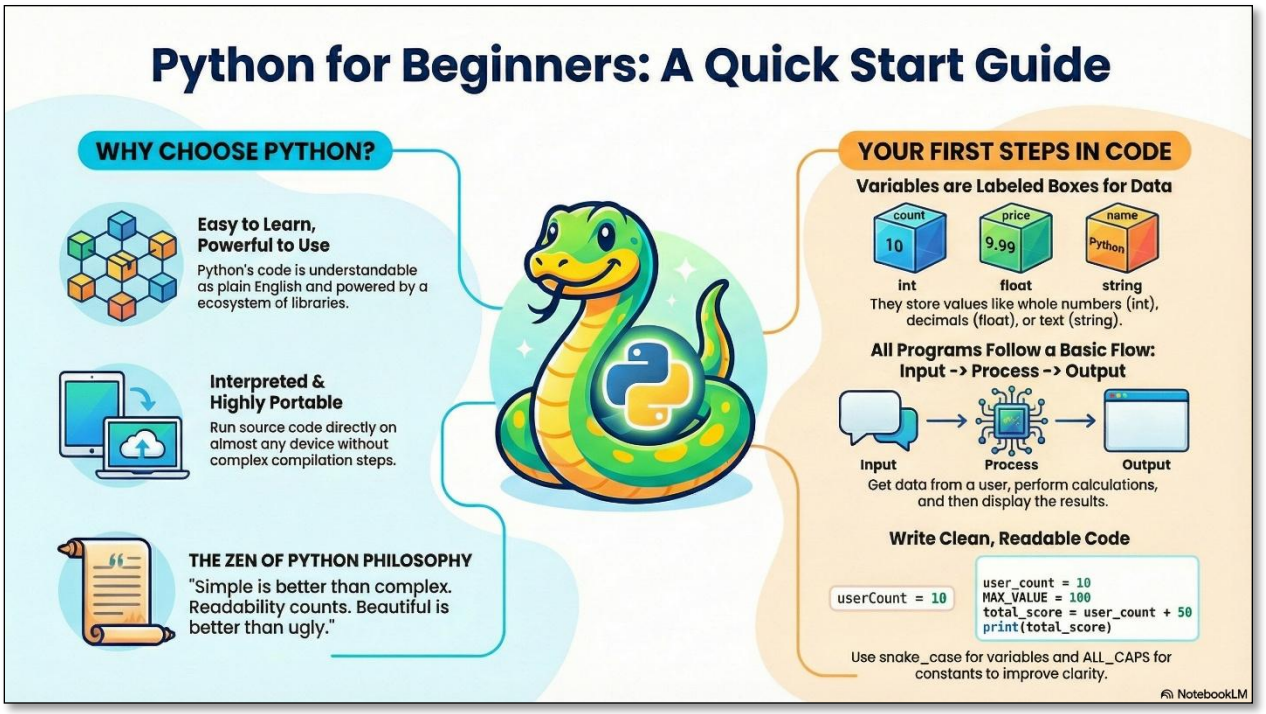
Chapter 2: Getting Started with Python

Contents



Red light: No AI

Time required: 180 minutes

NOTE: Please complete all tutorials and assignments.

Why Python?

Python's true power lies **not in the core language**, but in its **ecosystem of libraries**.

There are **hundreds of open-source Python libraries** that extend its capabilities into nearly every field, including:

- Image processing
- Networking
- Plotting and data visualization
- Web development
- Scientific and engineering computing
- Gaming
- Cryptography
- Databases
- GIS
- Audio and music
- Robotics and embedded devices
- Cybersecurity

It is hard to come up with a programming application area where someone has not already supplemented the basic Python programming language with an open-source library designed for use in that application area.

How Popular is Python?

https://pypl.github.io/PYPL.html

https://www.tiobe.com/tiobe-index/

The Story Behind the Name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Python Goals

In 1999, Guido van Rossum defined his goals for Python:

- an easy and intuitive language just as powerful as those of the major competitors;
- open source, so anyone can contribute to its development;
- code that is as understandable as plain English;
- suitable for everyday tasks, allowing for short development times.

Interpreted vs Compiled

In compiled languages like **C or C++**, the source code is translated into **binary code (0s and 1s)** using a **compiler**. This process includes setting flags and linking libraries.
When you run the program, a **linker/loader** loads it into memory and starts execution.

Python works differently.

Python does **not** require compilation to binary. You run the **source code directly**.
Behind the scenes, Python:

1. Converts the source code into **bytecode** (an intermediate form).
2. Translates the bytecode into the **computer's native language**.
3. Executes the program.

This process is automatic and makes Python much easier to use.
You don't have to compile, link libraries, or configure the build process.
This also makes Python programs **highly portable**—you can move your Python script to another computer, and it will likely run without changes.

Summary:

- **Compiled languages (like C++)** are faster but require more setup.
- **Python** is slower but easier to write and run across many different devices.

Free and Open Source

Python is an example of a FLOSS (Free/Libré and Open-Source Software). You can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like Kivy to create games for your computer and for iPhone, iPad, and Android.

Object-Oriented Programming

Python supports procedure-oriented programming as well as object-oriented programming. In procedure-oriented languages, the program is built around procedures or functions which are reusable pieces of programs. In object-oriented languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

The Zen of Python

Run the following code to find out about the Python mission. This includes good program design and coding principles.

At a Python prompt, type the following code. You should see the following The Zen of Python.

```
# Display the Zen of Python
import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Keyboard Shortcuts Save Time

The mouse is good for learning and exploring a program. To become a power user, keyboard shortcuts are essential.

The following keystrokes work in most Windows programs and can really speed up your work.

The most important keyboard shortcut is **CTRL+S** which saves the file. Get used to doing that on a very regular basis. Save early, Save often.

Keystroke	**Result**
CTRL+S	Save the file
CTRL+C	Copy selected text
CTRL+X	Cut selected text
CTRL+V	Paste
CTRL+Z	Undo the last keystroke or group of keystrokes
CTRL+SHIFT+Z	Redo the last keystroke or group of keystrokes
F5	Run module

How Does Typing in Code in a Tutorial Help with Learning?

Typing in code someone else created can significantly enhance learning in several ways:

- **Active Engagement**: Typing the code yourself forces you to actively engage with the material, rather than passively reading or watching. This active participation helps reinforce the concepts being taught.
- **Muscle Memory**: Repeatedly typing code helps build muscle memory, making it easier to recall syntax and structure when you write code independently.
- **Error Handling**: When you type code, you're likely to make mistakes. Debugging these errors helps you understand common pitfalls and how to resolve them, which is a crucial skill for any programmer.
- **Understanding**: Typing out code allows you to see how different parts of the code interact. This deeper understanding can help you apply similar concepts to different problems.
- **Retention**: Studies have shown that actively doing something helps with retention. By typing out the code, you're more likely to remember the concepts and techniques.

Practice, Practice, and More Practice - Neural Plasticity

Practice makes permanent.

This course will give you a lot of new information. To truly learn it, you need to **write code often**. Just like athletes or musicians don’t become experts overnight, programmers need **consistent practice** to build skill.

Your brain grows stronger with use. A little bit of programming each day strengthens the neural connections in your brain. This process is called **neural plasticity**—your brain literally rewires itself through repetition.

If you want to become a programmer, **spend time coding**, not just reading. Thinking, doing, and repeating forms strong mental habits.

Neural pathways are **strengthened** into habits through the repetition and practice of thinking, feeling, and acting.

PRACTICE: Start your morning passionately declaring aloud your goals for the day. Declarations send the power of your subconscious mind on a mission to find solutions to fulfill your goals.

Assignment 2.1: Organization and Folders

As we create Python programs, we want to be able to find them.

1. Create a folder named **Spring_26**
2. Under that folder, → create a folder named **Python**
3. Under Python → Create a folder for each week, Week_01, Week_02, etc.

Take a screenshot of your folder structure to include when submitting this assignment.

Tutorial 2.1: Hello World!

In this and future tutorials, you will **type and run sample programs**.

Learning to program means **writing code yourself**—not just reading it.

You are **not expected to understand everything right away**. The goal is to help you get familiar with what Python code looks like and how it works.

Hands-on practice is the key to learning.

Hello World is the traditional first program to create in any programming language. It confirms that your programming environment is setup properly.

When you open a folder with Visual Studio Code (VS Code), it automatically creates a project. This becomes important when we start working with more than one file.

1. In **File Explorer** → **Right Click** on your **Week 1** folder → **Show more options** → **Open with Code**
2. In **Visual Studio Code** → **File → New File** → Python file to create a new Python program file.
3. To save a Python program file, go to **File → Save As** → name the file **hello_world.py**
4. Click **Save**.

Let's extend the "Hello, World!" program you created earlier.

We'll write this program using the style and structure you'll follow throughout this course.

This helps build good habits from the start and prepares you for writing more complex programs later.

Let's get started!

At the top of your program, you'll add a **header**. This is a **multiline comment** that explains what the program does.

When Python runs the program, it **ignores comments**—they are only for people reading the code.

Use headers to describe the purpose of your program, the author, and the date.

```
"""
    Name: hello_world.py
    Author:
    Created:
    Purpose: My first Python program
"""
```

The **first line** is a **comment** that explains what the **next line does**.

Python uses the print() function to **display the contents of a variable** (or a message) on the screen.

```
# Print the literal string Hello World!
print("Hello Python World! Let's start coding!!")
```

Comments make your code easier to read and understand—for both you and others.

Example run:

```
Hello Python World! Let's start coding!!
```

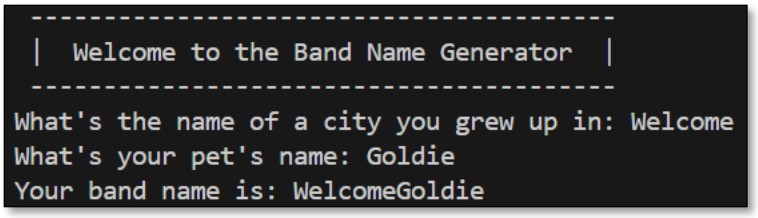
Tutorial 2.2: Band Name Generator

We will follow a step-by-step process to write a program that:

1. Takes **two string inputs** from the user.
2. Combines those inputs to create a **Band Name**.

This exercise will help you practice input, string handling, and output in Python.

Example run:



Development Process

In this tutorial, we will walk through the development process of a Python program. The goal of this program is to prompt the user for two separate string inputs and then combine these inputs to generate a creative band name. This exercise will help reinforce fundamental programming concepts such as user input, string manipulation, and output formatting.

Step-by-Step: Planning a Program

1. Define the Problem

Every program solves a problem. Start by clearly identifying what the program should do.

2. Create an Algorithm

An algorithm is a step-by-step plan to solve the problem. Think of it as the recipe for your program.

3. Write Pseudocode

Pseudocode is a simplified version of your algorithm written in plain language. It helps you plan the logic before writing actual code.

4. Benefits of Pseudocode
    - Focuses on logic, not syntax
    - Makes coding easier and faster
    - Helps catch mistakes early

Step-by-Step: Coding a Program

A well-planned program is clean, efficient, and easy to understand—like a well-crafted piece of art.

1. Create a Python program named: **band_name_generator.py**
2. Type in the following code.

This is a multiline comment that describes the program.

```
"""
    Name: band_name_generator.py
    Author:
    Created:
    Purpose: Get string inputs from user, concatenate them together
"""
```

Get two string inputs into two string variables from the user. A variable is a named storage location in dynamic memory or RAM.

```
# TODO: Ask the user for the city that they grew up in.
city = input("What's the name of a city you grew up in: ")

# TODO: Ask the user for the name of a pet.
pet = input("What's your pet's name: ")
```

String concatenation refers to the process of combining multiple strings into a single string. This can be achieved using the **+** operator.

```
# TODO: Combine the name of their city and pet.
band_name = city + pet
```

Display the band name using fstrings.

```
# TODO: Display their band name using fstrings
print(f"Your band name is: {band_name}")
```

Example run:

```
What's the name of a city you grew up in: Welcome
What's your pet's name: Goldie
Your band name is: Welcome Goldie
```

Understanding Variables

A key feature of any programming language is the ability to **use and manipulate variables**.

A **variable** is a name that refers to a value stored temporarily in the computer's memory.

You can **store, change, and use** these values throughout your program.

```
print("  Variable ")
print(" +---------+")
print(" | Address |")
print(" +---------+")
print(" |  Value  |")
print(" +---------+")
```

```
 Variable
+---------+
| Address |
+---------+
|  Value  |
+---------+
```

Variables as Labeled Boxes

Think of a **variable** as a **labeled box**.

- The **label** is the variable's name.
- You can **put something inside the box** (a value).
- When you want to use that value, you **look inside the box using its label**.
- You can also **replace what's inside** the box with something new anytime you wish.

This makes it easy to store and access information in your program.

In this course, we will generally use these three variable types:

- **int**: Whole numbers without decimals (e.g., 2, 100)
- **float**: Numbers with decimals (e.g., 2.23, 0.5)
- **string**: A sequence of characters or text (e.g., "hello", "Python123")

Tutorial 2.3: Inches to Centimeters

When you get input from the user using input(). Input into a computer always comes in as a **string**, even if the user types a number.

To use that input in math operations, you must **convert it to a number**:

- Use **int()** to convert to a **whole number**
- Use **float()** to convert to a **decimal number**

```
# Get string input from the user
inches = input("Enter inches: ")
# Convert a string to a float number like 3.114
inches = float(inches)
```

The float() function turns a string like "5.7" into the number 5.7, so you can use it in calculations.

In Python, we typically combine both of those operations in a single line.

```
# Get a string from the user, store a decimal number in the inches variable
inches = float(input("Enter inches: "))
```

1. Create a new Python program named **inches_to_cm.py**.
2. Copy and paste the following TODO code to get started with this program.

```
"""
    Name: inches_to_centimeters.py
    Author:
    Created:
    Purpose: Convert Inches to Centimeters
"""

# TODO: Print a nice title for our program

# TODO: Get inches input from user, cast to float

# TODO: Convert inches to centimeters

# TODO: Display the centimeter result
```

3. Type in the following code.

This is a multiline comment that gives information about the program. It is helpful to have a descriptive header in a program.

```
1  """
2      Name: inches_to_centimeters.py
3      Author:
4      Created:
5      Purpose: Convert Inches to Centimeters
6  """
```

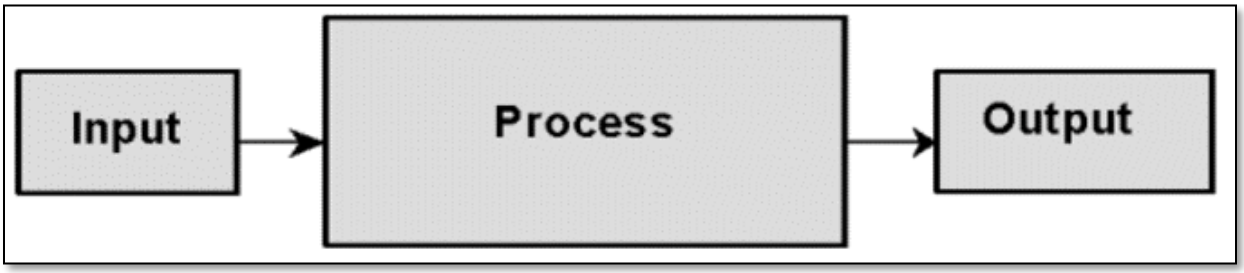
The first line is a single comment to let another programmer know what is going on.

The next lines print the program title to the console.

```
8   # TODO: Print a nice title for our program
9   print("-----------------------------------------")
10  print("|     Inches to Centimeters Converter     |")
11  print("-----------------------------------------")
```

Input - Process - Output

All programs have three basic components.



Input

```
13  # TODO: Get inches input from user, cast to float
14  inches = float(input("Enter inches: "))
```

- The first line is a comment that describes what the code is doing.
- The input() function displays the prompt and waits for the user to type something.
- The text in quotes is the prompt. It's a string that appears exactly as written.
- All input from the keyboard is a string, even if it looks like a number.
- We use the float() function to convert the input to a decimal number.
- The = operator assigns the converted number to the variable named inches.
- This line gets a decimal number from the user and stores it in the inches variable.

Process

The process section is where the program performs a calculation or processes data.

```
16  # TODO: Convert inches to centimeters
17  centimeters = inches * 2.54
```

- This line **converts inches to centimeters** using a standard formula.
- The = symbol is not an equal sign like in math—it's called the **assignment operator**.
- It means:
  Calculate the expression on the right.
  Store the result in the variable on the left.

In this case, the result of inches * 2.54 is stored in the centimeters variable.

Output

```
19    # TODO: Display the centimeter result
20    print(inches, "inches is equal to", centimeters, "centimeters.")
```

The last line uses the print function to print out the conversion result. Note the , (comma) sign between the variable names and the strings.

1. Run the program by pressing the F5 key or Click the run button.
2. The program will ask you to Enter inches. Type in 24 and press enter.

Example run:

```
----------------------------------------
|     Inches to Centimeters Converter    |
----------------------------------------
Enter inches: 24
24.0 inches is equal to 60.96 centimeters.
```

This program may seem short and simple—but that's the point.

Many websites perform similar conversions, and their code isn't much more complex than this.

Even though it looks small, this kind of program shows how real-world tasks can be solved with just a few lines of Python. Small, clear code can be powerful and useful.

Tutorial 2.4: Average

Requirements

This program will **compute the average of two numbers** entered by the user.

Steps:

1. Create a new Python program.
2. Save the file as **average.py**.

About Comments

Lines that start with # are called **comments**.

- Comments help **others understand** your code.
- They also help **you remember** what you were thinking when you wrote it.
- Writing clear comments is a **good programming habit**.

Title Block

```
"""
    Filename: average.py
    Author:
    Date:
    Purpose: Average two numbers input by user
"""
```

The Title Block in a program is part of documenting or commenting a program to make it more understandable. Please use this in each of your programs.

```
print("Calculate the average of two numbers.")
```

This is a description of the programs purposes for the end user.

Get Input

```
# Get input from user, convert string input to float
number1 = float(input("Enter the first number: "))
number2 = float(input("Enter the second number: "))
```

We need to get two numbers from the user. We get the numbers one at a time and assign each number to a variable.

Calculate

```
# Calculate average
average = (number1 + number2) / 2
```

When calculating the average, pay attention to **parentheses**:

- **Parentheses** change the **order of operations**.
- Without them, Python would divide number2 by 2 first, then add number1.
- With parentheses, Python adds first, then divides.

Python follows the **PEMDAS** rule:

P – Parentheses
E – Exponents
MD – Multiplication and Division (left to right)
AS – Addition and Subtraction (left to right)

Use parentheses to make your calculations correct and clear.

Display Output

```
17    # Display average
18    print(f"The average of {number1} and {number2} is: {average}")
```

The output of the program is shown to the user using a **formatted string**, called an **f-string**.

From this point on in the course, we will use **f-strings** to format output.

Example:

```
print(f"The average is {average}")
```

- f-strings make it easy to insert variables into a string.
- They also give you **control over how the data is displayed**, such as number of decimal places, alignment, and more.

Using f-strings makes your output **clear**, **professional**, and **easy to read**.

Example run:

```
Calculate the average of two numbers.
Enter the first number: 5
Enter the second number: 2
The average of 5.0 and 2.0 is: 3.5
```

Assignment 2.2: Age in Months

Objective:

Use input and basic math to convert age in years to months.

Instructions:

1. Ask the user to enter their **name**.
2. Ask for their **age in years**.
3. Calculate their **age in months**.
4. Display a message with their name and age in months.

Example run:

```
What is your name? Bill
How old are you? 69
Hello, Bill! You are 828 months old.
```

The Concept of Type

Strongly Typed Languages

One major difference between **Python** and languages like **Java** or **C++** is how they handle **data types**.

In **strongly typed languages**, such as Java and C++:

- Every variable must be **declared with a data type** before it is used.
- The data type tells the program what **kind of values** the variable can hold and what **operations** are allowed.

Example (Java):

```
int count = 10;  // 'int' means this variable holds whole numbers
```

This helps catch certain errors early but makes the language more rigid.

Python Is Not Strongly Typed

In contrast, **Python is dynamically typed**:

- You don't have to declare the data type.
- Python figures out the type **automatically** based on the value assigned.

Example:

```
count = 10      # Python sees this as an int
count = "ten"   # Now it becomes a string
```

This makes Python **easier to use**, especially for beginners.

However, this **flexibility comes at a cost**, such as **slower performance** and a **greater risk of bugs**.

Variable Declaration in Python

In **Java or C++**, you must declare variables before using them. In **Python**, you simply create a variable by assigning a value:

```
message = "Hello"
```

No need to specify a type.

Important Cautions

1. Don't Reuse Variable Names Unintentionally

In Python, if you assign a new value to a variable name, it **overwrites the old value**:

```
score = 95
score = "ninety-five"  # Now score is a string, not a number
```

This can lead to confusing bugs if you forget what type it's supposed to be.

2. Watch for Typos in Variable Names

Python **does not warn you** if you mistype a variable name:

```
total = 100
print(totla)  # Error: 'totla' is not defined
```

This kind of error can be **hard to find**, especially in longer programs.

Summary

- Strongly typed languages require explicit type declarations.
- Python is more flexible but requires **careful naming** and **attention to variable use**.
- Always use **clear, consistent names** and **comment your code** to avoid bugs.

Variable Names and Keywords

Programmers usually choose **meaningful variable names** that describe their purpose.

When naming variables in Python, follow these rules:

- **Do not use Python keywords** (reserved words like if, while, for) as variable names.
- Variable names **cannot contain spaces**.
- The **first character** must be a letter (a–z, A–Z) or an underscore (_).
- After the first character, you can use letters, digits (0–9), or underscores.
- Variable names are **case sensitive**:
  Number and number are two different variables.
- Variable names can be long and may include letters and numbers, but **cannot start with a number**.

snake_case Variable Names

In Python, variables often combine multiple words using the **snake_case** convention:

- Use **all lowercase letters**.
- Separate words with **underscores (_)**.

Examples of snake_case:

- gross_pay
- hours_worked
- total_hours_worked
- user_name
- average_score

Using snake_case improves code readability and follows Python community standards.

Examples of Illegal Variable Names and Errors:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax

>>> more@ = 1000000
SyntaxError: invalid syntax

>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

Why are these illegal?

- 76trombones starts with a **number** — not allowed.
- more@ contains an **illegal character** (@).
- class is a **Python keyword** and cannot be used as a variable name.

Python Keywords

Keywords are reserved words that define the structure and syntax of Python programs. You **cannot** use these as variable names.

Python reserves 35 keywords:

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	async
def	for	lambda	return	await

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

Why Use Constants?

Why Avoid "Magic Numbers"?

- Using numbers directly in code without explanation is called using **magic numbers**.
- For example, 2 could mean anything — 2 cats, 2 percent, 2 hours.
- Magic numbers make code hard to understand and maintain.

Use Named Constants

- Replace magic numbers with constants that have meaningful names.
- If you need to change the value, you only update it in one place.
- This makes your code easier to read and maintain.

Naming Convention for Constants

- Constants are named with **all capital letters**.
- Words are separated by **underscores (_)**.

```
A_SIMPLE_CONSTANT = 10
ANOTHER_CONSTANT = 3.14
SALES_TAX = 0.07
```

Math Operators and Operands

Symbol	Operation	Example	Description
+	Addition	a + b	Adds two numbers
–	Subtraction	a – b	Subtracts one number from another
-	Negation	- a	Change sign of operand
*	Multiplication	a * b	Multiplies one number by another
/	Division	a / b	Divides one number by another and gives the result as a floating-point number
//	Integer division	a // b	Divides one number by another and gives the result as a whole number
%	Remainder or Modulus	a % b	Divides one number by another and gives the remainder
**	Exponent	a ** b	Raises a number to a power

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

The operators +, -, *, /, and ** perform addition, subtraction, multiplication, division, and exponentiation, as shown in the following examples.

```
20 + 32
hour - 1
hour * 60 + minute
minute / 60
5**2 (5 + 9) * (15 - 7)
```

The result of any number division in Python is a floating-point result as shown below.

```
minute = 59
minute / 60
# Output: 0.9833333333333333
```

In the following example of integer division, the value is truncated, the fractional part is left out.

```
5 // 2
# Output: 2
```

The Newline Character (\n)

- **\n** is an escape sequence that represents the newline character.
- It tells the interpreter to **start a new line** when printing a string.
- Though **invisible** like wind, its effect is **visible** in the output format**.**

```
print("Dick\nBaldwin")
# Output: Dick
# Output: Baldwin
```

- In the example, \n causes the last name to appear on the next line.
- To make newlines visible in a string, we use the \n notation.

Tutorial 2.5 Savings Account

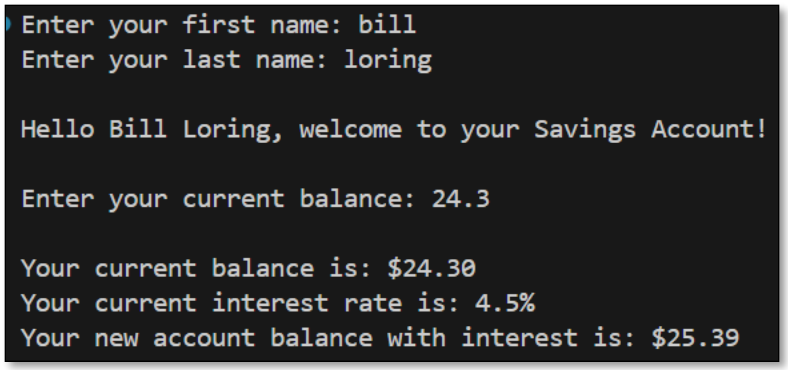
Requirements

Create a program that calculates a savings account balance using a constant and other concepts we have learned.

1. Save the program as **savings_account.py**.

```python
"""
Name: savings_account.py
Author:
Date:
Purpose: Demonstrate the use of constants, user input,
and basic arithmetic operations
"""

# TODO: Declare interest rate constant
INTEREST_RATE = 0.045

# TODO: Get user first and last name
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")

# TODO: Use string methods to capitalize first and last name
first_name = first_name.title()
last_name = last_name.title()

# TODO: Concatenate first and last name
full_name = first_name + " " + last_name

# TODO: Get floating point input from user
print(f"\nHello {full_name}, welcome to your Savings Account!")
balance = float(input("\nEnter your current balance: "))

# TODO: Calculate new balance
new_balance = (balance * INTEREST_RATE) + balance

# TODO Display your current interest, start by echoing user input
print(f"\nYour current balance is: ${balance:,.2f}")
print(f"Your current interest rate is: {INTEREST_RATE * 100}%")
print(f"Your new account balance with interest is: ${new_balance:,.2f}")
```

Example run:



Choosing Mnemonic (Easy to Remember) Variable Names

Key Rules for Variable Names in Python:

- Can use **letters**, **numbers**, and **underscores**.
- Cannot contain **spaces**.
- Cannot start with a **number**.
- Are case-sensitive (**rate**, **Rate**, and **RATE** are different).

Why Mnemonic Names Matter:

Python doesn't care what you name variables—as long as the names follow the rules.

```
a = 35.0
b = 12.50
c = a * b
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

All 3 do the same thing. But only the second is **clear to the reader**.

Mnemonic = Memory Aid

Good variable names describe their **purpose** or **meaning**.

Assignment 2.3: Susie's Square Calculator

Let's finish off this chapter by calculating the area and perimeter of a square. It is time for your to create your first program.

- Formula for the area of a square:

$$A = s^2$$

  where **A** is the area and **s** is the length of a side.

- Formula for the perimeter of a square:

$$P = 4s$$

  where **P** is the perimeter and **s** is the length of a side.

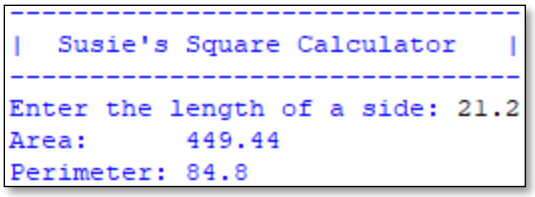
Create a Python program named **square_calculator.py**

You can pseudocode your code by using the **TODO** list shown below. This allows you to plan and start commenting your code.

1. Copy and paste the following to start your program.

```
"""
    Name: square_calculator.py
    Author: William A Loring
    Created: 07/08/22
    Purpose: Calculate the area and perimeter of a square
"""

# TODO: Print a creative title for our program

# TODO: Get length of a side from the user, convert to float

# TODO: Calculate area

# TODO: Calculate perimeter

# TODO: Display results
```

Example run:



Glossary

assignment A statement that assigns a value to a variable.

concatenate To join two operands end to end. Typically used with strings and string literals.

comment Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

evaluate To simplify an expression by performing the operations in order to yield a single value.

expression A combination of variables, operators, and values that represents a single result value.

floating point A type that represents numbers with fractional parts.

integer A type that represents whole numbers.

keyword A reserved word that is used by the compiler to parse a program; you cannot use keywords like if, def, and while as variable names.

mnemonic A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

modulus operator An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.

operand One of the values on which an operator operates.

operator A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

statement A section of code that represents a command or action. So far, the statements we have seen are assignments and print expression statement.

string A type that represents sequences of characters.

type A category of values. The types we have seen so far are integers (type int), floating-point numbers (type float), and strings (type str).

value One of the basic units of data, like a number or string, that a program manipulates.

variable A name that refers to a value.

Assignment Submission

1. Attach all tutorials and assignment code files.
2. Attach screenshots showing the successful operation of each tutorial program.
3. Attach a screenshot showing your folder structure with your Python programs for week 1.
4. Submit in Blackboard.