Annotated Bibliography

“Python 3.14.3 Documentation.” *Python.org*, Python Software Foundation, https://docs.python.org/3/.

This source is the official documentation for Python, created and maintained by the Python software base Foundation. I selected this source because it is the most accurate reference for Python;s syntax, built-in function, libraries and language rules. It provides a detailed explanation of every feature, from basic data type to advanced modules, which will help ensure that information in my guide will be correct and up to date. This source is useful because it includes examples, definitions, and technical explanations that I can break down into simpler language for beginners. One limitation is that the documentation can be overwhelming and difficult for me since I'm rather new, so I will have to find a way to translate the complex idea into more reasonable explanations.

PyCoder. “The Evolution and History of Python: From Hobby Project to Global Dominance (1989–2026).” *PyCoder*, 5 Jan. 2026.

This source provides a historical overview of Python’s development from its creation to its modern day use. I selected this source because by understanding the history of Python it can help explain why the language is designed the way it is and why it became so popular today. The source is useful because it highlights major version changes, it provides sample close, and the idea behind Python’s simplicity, and how the language expanded into fields like data science and web development. This background will help strengthen the introduction of my guide and give the reader control about Python’s purpose. One flaw is that the article focuses more on history rather than technical details so I will need additional sources for deeper explanation of syntax and features.

“History of Python.” *Wikipedia*, Wikimedia Foundation, https://en.wikipedia.org/wiki/History_of_Python

This wikipedia page summarizes Python’s origins, its creator Guido van Rossum, and the evolution of its major version like before. I think this source will be cool because it provides a broad overview that is easy to understand and useful for cross-checking facts about technical sources. It helped me understand Python's early stages, such as readability, which are an important theme. I found this source useful because it has a timeline, reference and links to related topics that can support my research. A downside is that wikipedia pages can be edited by anyone so I cannot rely on it as a primary source. Instead I will use it to confirm information.

CodeRivers. “The History of the Python Language: A Journey Through Time.” *CodeRivers*, 22 Apr. 2025.

This article explains more about Python's history once again, but this time it focuses on how the language grew from a small project into a widely used tool. This source might be useful since it presents Python's development in a clear and engaging way, making it easier to understand the motivation behind its design. It also discusses best practices coding style which will help for me to write my section of my guide about writing readable code. One flaw is that it does not in-depth into technical features so I will mainly use it for background information and writing style guidance.

"PythonDocs – Comprehensive Python Documentation." *PythonDocs.org*, https://pythondocs.org/.

This website is an unofficial documentation site that organizes Python concepts in a simpler more beginner friendly format. I selected this because it breaks down complex topics into short explanations and examples, which will help me structure my guide in a way that is easy for new learners to follow. This source is useful because it includes step by step tutorials, diagrams, and explanations of common beginner mistakes. It also provides alternative explanations that complement the official documentation. One limitation is the precision it has compared to the official Python site, so I will have to cross reference.

Eric Python Crash Course
Matthes, Eric. *Python Crash Course*. No Starch Press, 2019.

This book provides a structured introduction to Python, converting basic syntax, data structures, and beginner projects. I selected this source because it mirrors the structure I want for my guide: starting with fundamentals and gradually moving into advanced topics. It is useful because it includes clear explanations, diagrams, and hands on exercises that help reinforce learning. I found this source helpful for me in understanding the Python concept in a logical order. One drawback is that the book's example may be too long or detailed for my guide.

Hamedani, Mosh. "Python Tutorial – Python for Beginners." *YouTube*, uploaded by Programming with Mosh, https://www.youtube.com/watch?v=kqtD5dpn9C8

This youtube video is a full beginner-friendly Python course created by Mosh, he is pretty clear throughout the video. So selected this source because it provides a complete walkthrough of Python basics in a single organized video. The video tutorials cover variables, loops, functions, all these align well with the beginner section of my guide. I found this source useful since it explains concepts visually and includes some examples that I can analyze and rewrite in my own world. One drawback is that it doesn't cover everything so I have to do additional research for more beginner concepts.

Bro Code. “Python Full Course for Beginners.” *YouTube*, uploaded by Bro Code, 3 Jan. 2023, https://www.youtube.com/watch?v=ix9cRaBkVe0..

This YouTube video is a full beginner Python course made by Bro Code, he knows to be clear, he also has content on other coding languages like C++. I selected this source because he is reliable and his teaching style is structured, and straightforward. It is similar to Jenkov java website tutorials, which is the model I want to follow for my own Python guide. The video covers all the essential Python concepts (12 hours) such as variables, data types, loops, functions, lists, dictionaries, and basic file handling, making it a really strong foundation for my guide. I found this source useful because it demonstrates each concept with live code examples, which help me reinforce the journey for Python. One drawback is that it is really quick even though it is 12 hours and there are so many concepts and the Youtube limit is 12 hours so he has to cram a lot of concepts.

“Python Tutorial.” *W3Schools*, Refsnes Data, https://www.w3schools.com/python/..

This source is an online Python tutorial from W3Schools, they are a really well-known website that provides tons of beginner friendly programming lessons and examples. I selected this source because this teaching style is simple and structured the way I want to model my own Python guide. The tutorial covers a wide range of topics, including basic syntax, variables, data types, loops, functions, list, and modules. I found this source useful because each concept is explained well with short code snippets that make it understandable and practical. It also includes interactive examples that allow me to test directly in the browser, which can help my learning process. One drawback is that W3Schools oversimplifies so I have to cross reference with my other source.

NeuralNine. “Python Tutorial for Beginners (Full Course).” *YouTube*, uploaded by NeuralNine, 14 Feb. 2021, https://www.youtube.com/watch?v=H2EJuAcrZYU..

This another YouTube video that is really long and goes indepth. The channel provides clear explanations, and programming tutorials. I choose this source because the person teaches in a step by step style that is easy to follow. The video covers the essentials like I said before so I can use it to cross reference my previous sources. It also talks about object oriented programming which is something that will be in the intermediate area. I found this source useful because it demonstrates a lot of concepts through live coding, which is really helpful for me to understand how the base syntax works. The pacing is pretty beginner friendly and examples are simple enough for me to write in my own words. One drawback is that video doesn’t go into advanced concepts, so I should focus on the foundation piece then do more research later for high level concepts.

Introduction

Python Introduction

What is Python?

Python is a programming language designed to be simple, readable, and easy to learn. It allows you to write clear, concise programs without worrying about complex syntax. Python is used in many areas, including web development, automation, data analysis, artificial intelligence, and general scripting.

Why Learn Python?

So Python is one of the most beginner-friendly languages because its syntax is close to English. This makes it easier to focus on learning programming concepts instead of getting stuck on punctuation or formatting. Python has a large ecosystem of libraries, which means you can build almost anything with it.

How Will This Guide Work

This guide will introduce Python step by step, starting with basic concepts and gradually moving into more advanced topics. Each section will include short explanations and practical examples, similar to the structure of Jenkov's Java tutorials. The goal is to help you understand Python and build confidence as you learn.

Who This Guide Is For

This guide is designed for beginners who want to learn Python from scratch, as well as anyone who prefers simple explanations supported by small code examples.

Syntax

What is Python Syntax

Python syntax refers to the set of rules that defines how Python code is written and interpreted. Every programming language has its own structure and Python is known for having one of the simplest and most readable syntaxes. Understanding Python's basic syntax is essential before learning about other topics such as variables, or any other concept.

Python highlights clarity and consistency. Instead of using symbols like {} or ; to structure code, Python uses indentation and line breaks. This makes Python code look clean and easier to follow compared to most programming languages.

Python Code Structure

Python uses indentation instead of braces in many languages, code blocks are wrapped in {}. Python uses indentation (spaces or tabs) to define blocks of code.

Example:

```
if True:
    print("This line is inside the block")
print("This line is outside the block")
```

The indented line belongs to the **if** block
The unindented lines do not.

Python requires consistent indentation. Most people use 4 spaces per indentation level.

Python Does Not use Semicolons

In languages like Java or C++, every statement ends with a semicolon. Python on the other hand ends the statement with a **newline**, not a semicolon.

Example:

```
x = 10
y = 20
print(x + y)
```

Semicolons can be used, but they are almost never needed.

Python Is Case-Sensitive

Variable and function names must match exactly

```
name = "Alex"
Name = "Sam"
```

These are two different variables.

Comments in Python

For **Single-Line Comments** use **#** to write comments.

```
# This is a comment
x = 10  # This is also a comment
```

For **Multi-Line Comments Python** itself does not have a direct multi-line comment symbol, but triple quotes are commonly used:

```
"""
This is a multi-line comment.
It is often used for documentation.
"""
```

Python Statements

A statement is a single instruction.

```
x = 5
print("Hello")
if x > 3:
    print("x is greater than 3")
```

Each line is a separate statement unless you use a line continuation.

Line Continuation

Python allows breaking lines inside parentheses, brackets, or braces.

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

Explicit Line Continuation

We will use a backslash here \:

```
total = 1 + 2 + 3 + \
        4 + 5 + 6
```

General Whitespace Rules

Python treats whitespace really meaningfully.

Correct:

```
if x > 0:
    print("Positive")
```

Incorrect:

```
f x > 0:
print("Positive")   # Error: indentation missing
```

Indentation errors are one of the most common beginner mistakes.

Python Keywords

Keywords are reserved words that cannot be used as a variable names:
Examples includes:

- if
- else
- for
- while
- class
- def
- return
- import
- True, False, None

Trying to use a keyword as a variable will cause an error.

Basic Print Syntax

The print() function is one of the first things beginners learn.
Example:

```
print("Hello, world!")
```

You can also print variables, numbers or expressions
Example:

```
x = 10
print("x =", x)
```

Python Input Syntax

To get user input:

```
name = input("Enter your name: ")
print("Hello,", name)
```

All inputs are stored as a string unless converted.

Expressions and Operators:

Python supports arithmetic expressions:
Example:

```
x = 10
print(x > 5)   # True
```

These expressions follow standard math rules.

Common Syntax Errors:

Missing Indentation

python
```
if x > 0:
print("Positive")   # Error
```

Missing Parentheses

python
```
print "Hello"   # Error in Python 3
```

Using the wrong quotes

python
```
message = 'Hello"   # Error
```

Forgetting a colon

python
```
if x > 0   # Error
   print("Positive")
```

Mini-Excercises:

1.) Write a Python program that prints two lines of text.
2.) Create a block of code using if and indentation.
3.) Write a comment explaining what your program does.
4.) Break a long mathematical expression into multiple lines.

Solutions

Write a Python program that prints two lines of text.
Example:

```
print("Hello")
print("Welcome to Python")
```

Create a block of code using if and indentation.
Example:

```
x = 5
if x > 0:
    print("x is positive")
```

Write a comment explaining what your program does.
Example:

```
# This program prints a greeting
print("Hello, world!")
```

Break a long mathematical expression into multiple lines.
Example:

```
total = 1 + 2 + 3 + \
        4 + 5 + 6
print(total)
```

Variables

What Are Variables

Variables in Python are names that store values. When you assign a value to a name, Python automatically creates the variable for you. Unlike many other programming languages, Python does not require you to declare the type of a variable before using it. The type is determined by the value assigned.

Variables are essential because they allow programs to store information, reuse data, and also perform calculations.

Creating Variables

A variable is created when you assign a value using the **=** operator.
Example:

```
x = 10
name = "Minghao"
pi = 3.14
```

Explanation:

x stores an integer
name stores a string
pi stores a float number
Python figures out the type automatically.

Reassigning Variables

Python allows variables to change type any time.
Example:

```
value = 5
value = "five"
```

Explanation:

The same variable name can hold different types of data. This flexibility makes Python easy to use, but it also means you must keep track of what type your variables currently hold.

Multiple Assignments

Python supports assigning multiple variables in one line.

Example:
a, b, c = 1, 2, 3

You can also assign the same value to multiple variables:

x = y = z = 0

Variable Naming Rules

Python has specific rules for naming variables:

- Must start letter or underscore
- Cannot start with a number
- Cannot contain spaces
- Cannot use character like @, $, %
- Case sensitive (age, Age, and AGE are all different)

Python developers follow the snake_case:

user_name = “Alex”
total_score = 95

Dynamic Typing

Python is a dynamically typed language.
This means:

- You don’t declare types
- Types can change at runtime
- Variables are reference to objects in memory

Example:
x = 10
x = x + 5
x = "Now I'm a string"

Checking Variable Types:

You can use the type() function to check what type a variable currently holds.
Example:

```
x = 42
print(type(x))
x = "hello"
print(type(x))
```

Output:
<class 'int'>
<class 'str'>

Deleting Variables:

You can delete a variable using del.
Example:

```
x = 10
del x
```

Trying to use x after deletion will cause an error.

Variable and Memory:

In Python:

- Variables are names
- Values are objects stored in memory
- Variables point to objects

Example:

```
a = 10
b = a
```

Here both a and b point to the same value so changing a does not change b unless the value is mutable. (Like lists)

Mutable vs. Immutable Variables

Some data types can be changed (mutable), while others cannot (immutable)

Immutable Types:

- int
- float
- str
- tuple

Mutable Types:

- list
- dict
- set

Mini-Exercises:

1.) Create three variables: one integer, one string, and one float.
2.) Reassign a variable to a different type.
3.) Assign the same value to three variables in one line.
4.) Use type() to check the type of a variable.
5.) Delete a variable and try printing it
6.) Create two variables that reference the same value, then change one and see what happens.

Methods:

Create three variables: one integer, one string, and one float.
```
age = 18
name = "Adil"
height = 6.6
```

Reassign a variable to a different type.
```
x = 10
x = "ten"
```

Assign the same value to three variables in one line.
```
a = b = c = 0
```

Use type() to check the type of a variable.
```
x = 3.14
print(type(x))
```

Delete a variable and try printing it
```
x = 5
del x
# print(x)  # This will cause an error
```

Create two variables that reference the same value, then change one and see what happens.
```
a = [1, 2, 3]
b = a
a.append(4)
print(b)   # b also changes because lists are mutable
```

Data Types

Data Types Overview

Python's data types define the kind of values a program can work with. Unlike languages like Java, Python does not separate primitive types from object types. Every value in Python is an object, including integers, booleans, and even function. This means all values have methods, have identity, and are referenced by variables.
Understanding how Python stores and manages these objects is essential for writing a correct program.

Object Mode; Everything is an Object

Again in Java, int and Integer behave differently.
For example in Python you can have
x = 10
This indicates that 10 is already a full object of type int.
This means:

- Python does not need auto-boxing
- Python does not have primitive types
- All values behave consistently
- Variables are references, and not containers.

So when you assign a value to a variable you are binding a name to an object.

Identity, Type, and Value

Every Python object has three core properties:

1.) Identity: This is where the object lives in memory
2.) Type: What kind of object it is
3.) Value: The data stored inside the object

Example for inspection:

```
x = 42
print(id(x))     # identity
print(type(x))   # type
print(x)         # value
```

Identity never changes for an Object. While the value may or may not change depending on mutability.

Immutability vs. Mutability

Immutable Type: These cannot be changed after creation

- int
- float
- bool
- str
- tuple
- frozenset

So when you modify an immutable value Python will create a new object.
Example:

```
x = 10
print(id(x))

x = x + 1
print(id(x))
```

The identity changes meaning a new object was created.

Mutable Types: These can be changed in place

- list
- dict
- set

Example:

```
a = [1, 2, 3]
print(id(a))

a.append(4)
print(id(a))
```

The identity stays the same while the object itself changes.

Reference Behavior:

Variables in Python do not store values; they reference objects.
Example:
python
a = [1, 2, 3]
b = a

Both a and b reference the same list

python
a.append(4)
print(b) # [1, 2, 3, 4]

Data Type Categories:

Numeric Type:

Integers(int)

- Unlimited size
- Immutable
- Support arithmetic, bitwise operation, and conversations

Float(float)

- More precision (decimals)
- Subject to round error

Complex Numbers(complex)

Python supports complex numbers natively:
Example:
z = 3 + 4j

Boolean(bool)

Booleans are a subclass of integers

True == 1
False == 0

This will allow boolean values to participate in arithmetic.

String(str)

String are immutable sequences of characters.
s = “hello”
t = s.upper()

t is a new string while s is unchanged.

Sequences:

Lists

Mutable, ordered collections.

Tuples

Immutable, ordered collections

Ranges

Efficient sequences of integers

Mappings

Dictionaries(dict)

Key-value pairs.
Key must be immutable types.

Sets

Unordered collection of unique values.

NoneType

Represents “no value.”

IT is equivalent to Java’s null.

Type Conversion (Casting)

Python supports both implicit and explicit conversions.

Implicit Conversion: This occurs in mixed arithmetic

```
result = 5 + 2.0   # becomes float
```

Explicit Conversion:

```
int("10")
float("3.14")
str(42)
list("abc")
```

Custom Data Types

Just like Java allows custom classes, Python let you create your own types using classes.

Example:

```
class Person:
    pass
```

This instance of custom classes behaves like any other Python object.

Exercises to try:

These exercises are exploratory. There is no single correct answer so try running the examples and observing how Python behaves.

1.) Show that integers are immutable by comparing object IDs before and after modification.
2.) Demonstrate the difference between == and is.
3.) Convert a float to a string and back.
4.) Create a custom class and instantiate it.
5.) Show an example of implicit type conversion.

Operators

Python Operators

Operators in Python are symbols or keywords that perform operations on values. They are used to build expressions, compare data, combine conditions, and manipulate variables. Python's operator system is designed to be readable and consistent. And by understanding how each operator behaves it is essential before moving forward.

Arithmetic Operators

So Arithmetic operators perform mathematical operations on numeric values. Python supports addition, subtraction, multiplication, division, floor division, modulus, and exponentiation. Unlike some languages, Python's division operation always produces a floating point result, even when dividing two integers. Floor division // is useful when you need whole number results, such as in indexing or splitting data.

Example:

```
a = 10
b = 3

print(a + b)
print(a / b)
print(a // b)
print(a % b)
print(a ** b)
```

Note:

Python automatically promotes integers to floats when needed, which prevents overflow issues.

Comparison Operators

Comparison operators evaluate relationships between values and return True or False. These operators are used often in conditions and loops. Python also supports chained comparison allowing expressions like 1 < x < 7 to be written naturally. This chaining is evaluated left to right and stops early if a comparison fails.

Example:

```
x = 5
print(1 < x < 10)
```

Common Pitfall:
Comparing incompatible types like a string and an integer will raise a TypeError, unlike older versions of Python which allowed for arbitrary comparison.

Logical Operators

Logical operators combine boolean expressions. Python uses the keywords and, or, and not, which read more naturally than symbols like && or ||. Python evaluates logical expressions using short circuit evaluation, meaning it stops as soon as the final result is known. This can prevent unnecessary work or even avoid errors.

Example:
```
print(True or (1 / 0))   # No error because OR short-circuits
```

Importance:
Short-circuiting allows logical operators to be used safely in conditions that might otherwise cause exceptions.

Assignment Operators:

Assignment operators bind names to objects. The basic operator = assigns a value, while compound operator like += and *= update the variable based on its current value. These operators do not modify immutable objects; instead, they create new ones. For mutable objects, compound assignment may modify the object in place.

Example:
```
x = 10
x += 5
```

Behavior example:
```
a = [1, 2]
b = a
a += [3]   # modifies list in place
print(b)   # b also changes
```

Membership Operators:

Membership operators test whether a value exists within a sequence or collection. Python provides **in** and **not in**, which work with strings, lists, tuples, set, and dictionaries. For dictionaries, membership check keys by default, and not values.

Example:
```
print("a" in "cat")
print(2 in [1, 2, 3])
```

Note:
Membership checks on sets and dictionaries are really fast because of hash-based lookup.

Identity Operators

Identity operators check whether two variables reference the same object in memory. Python uses **is** and **is not** for identity checks. This is different from **==**, which checks for value equality. Identity checks are most commonly used with **None**, since **None** is a singleton.

Example:
x = None
print(x is None)

Common Pitfall:
Two lists with the same contents are equal but not identical.

Bitwise Operators:

Bitwise operators manipulate integers at the binary level. THese include AND(**&**), OR(**|**), XOR (**^**), NOT(**~**) and but shifts(<<, >>). While not used often in everyday Python programming, they are essential in tasks involving flags, compression, encryption, and low data manipulation.

```
x = 6   # 110
y = 3   # 011

print(x & y)   # 010
print(x | y)   # 111
```

Importance:
Bitwise operations allow Python to interact with binary data efficiently, which is important in networking, graphics, and system programming.

Mini-Exercises

1.) Use arithmetic operators to compute the area of a rectangle and the circumference of a circle.
2.) Write a condition using comparison and logical operator to check if a number is between 10 and 20 and is seven.
3.) Use assignment operators to update a score variable through several operations.
4.) Demonstrate the difference between == and is using lists.

Input & Output

Python Input and Output

Input and output are the primary way a Python program communicates with the outside. Output allows your program to display information to the user, while input allows the user to provide data back into the program. Python keeps these operations simple, but there is important behavior beneath the surface such as type conversion, formatting rules, and buffering that will affect how input and output work in real programs.

Output in Python

The print() function is Python's standard output function. It sends text to the console and wil automatically add a newline at the end unless told otherwise. Python converts the arguments to strings before printing them, which means you can print numbers, booleans, lists, and even custom objects without manually converting them.

Example:
print("Hello, world!")
print(67)
print(True)

Behavior Note:
If you print multiple values, Python separates them with a space by default. This can be changed using the sep parameter.

Customizing Output

Python's print() function supports optional parameters that control formatting:

- sep: This changes the separator between values.
- end: Changes what is printed at the end of the line.
- file: This sends output somewhere other than the console.
- flush: This forces immediate output.

Example:
print("A", "B", "C", sep="-", end="!")

Importances:
These options allow you to format output clearly without string concatenation, which keeps code readable and efficient.

String Formatting

String formatting allows you to combine text with variable, expressions and computed values in a readable way. Python provides several formatting systems, but modern Python favors f-strings because they are fast. Formatting is essential for producing clean output, especially when displaying numbers, aligning text, or embedding values inside messages. Understanding how formatting works helps you avoid messy concatenation and makes your program easier to maintain.

f -Strings

F-string allows you to embed variables and expressions directly inside a string using {}. Python evaluates everything inside the braces at runtime, which means you can insert calculations, function calls, or even conditional expressions. This makes f-strings both powerful and readable, and they are the preferred formatting method in modern Python.

Example:
```
name = "Alex"
print(f"Hello, {name}")

print(f"2 + 3 = {2 + 3}")
```

Behavior Note

Because f-strings evaluate expressions, they can replace many temporary variables and reduce clutter in your code.

format() Method

Before f-string the format() method was the standard way to insert values into strings. It uses placeholders {} that are replaced by arguments passed to the methods. While still widely supported, f-string is generally more recommended for beginners.

Example:
```
name = "Alex"
print("Hello, {}".format(name))
```

Importance

Understanding format() helps when reading older Python code or working with libraries that still use it.

Formatting Numbers and Alignment

Python's formatting tool allows you to control decimal places, alignment, padding, and more. This is really useful when printing tables, reports.

Example:
value = 3.14159
print(f"{value:.2f}") # 3.14
print(f"{value:>10}") # right-aligned

Behavior
Formatting does not change the underlying value; it will only affect how the value is displayed.

Input() Function

The input() function is Python's primary way of receiving information from the user. When called it pauses the program and waits for the user to type something and press Enter. Whatever the user typed is returned as a string, regardless of whether it is a number, boolean, or any other type. This design keeps input simple, but it also means programmers must convert the input manually if a different type is needed.

Example:
name = input("Enter your name: ")
print("Hello,", name)

Note:
input(0 always returns a full line of text, including spaces. If the user presses Enter without typing anything, the result is an empty string (“”).

Converting Input Types

Input() always returns a string, type conversion is necessary whenever you want to work with numbers or other data types. Python provides built-in functions like int(), float(), and bool() to convert strings into other types. However, these conversions only work if the input is valid; otherwise, Python will raise a ValueError.

Example:
age = int(input("Enter your age: "))
height = float(input("Enter your height: "))

Importance:
The program must assume the user will enter unexpected or invalid data. So in practice, input conversion is often wrapped in try/except blocks to prevent crashes and provide helpful error messages.

Input Buffering

Python uses a buffered input system, meaning it waits for the user to press Enter before returning any data. The entire line of text is stored in memory as a single string. This behavior has several drawbacks:

- Input is not returned character by character
- Escape characters are interpreted literally
- The program cannot continue until the user submits the line

Example:
```
text = input("Type something: ")
print("You typed:", repr(text))
```

Note:
Using repr() reveals hidden characters, making it easier to debug input issues such as accidental spaces or invisible characters.

Combine Input and Output:

Most programs combine both input and output to create a conversation with the user. Output provides instructions or feedback, while input collects information needed for the next step. This pattern forms the basis of command line tools, text based games, quizzes, and data entry programs.

Example:
```
name = input("What is your name? ")
print(f"Nice to meet you, {name}.")
```

Why This Matter
By understanding how input and output work together, you can design programs that guide the user clearly and respond dynamically to their actions.

Mini-Excercise to try

1.) Ask the user for their name and print a personalized greeting.
2.) Read two numbers from the user, convert them to integers, and print their sum.
3.) Demonstrate what happens when invalid input is passed to `int()`.
4.) Use `repr()` to show the exact characters the user typed.
5.) Create a small program that asks for a temperature in Celsius and prints the Fahrenheit equivalent.

Control Flow

Control Flow In Python

Control flow basically determines the order in which a program's statements are executed. Without control flow, a program would simply run from top to bottom with no ability to make decisions or react to different conditions. Python provides several control flow structures that range from **if**, **elif**, and **else** that allow programs to choose between multiple paths based on the values it encounters. Understanding control flow is essential before learning loops, functions, or other types of program logic.

If Statement

The **if** statement is the foundation of decision-making in Python. It evaluates a condition, and if the condition is true, the indented block beneath it will execute. Python relies on indentation rather than braces to define the block, which makes the structure visually clear. Conditions can be simple comparisons or complex expressions involving multiple operators.

Example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Note:
Python treats any non-zero number, non-empty string, or non-empty collection as True when used in a condition. Empty values evaluate to False, which allows for checks.

elif Clause

The **elif** clause allows you to test additional conditions when the initial **if** condition is false.Python evaluates each condition in order and executes the first block whose condition is true. Once a matching condition is found, the remaining **elif** and **else** blocks are skipped. THis structure prevents deep **if** statements and keeps the logic readable.

Example:

```
score = 85

if score >= 90:
    print("A")
elif score >= 80:
    print("B")
```

Importance:
elif creates clean, linear decision chains that are easier to maintain than multiple conditions.

else Clause:

The **else** clause provides a fallback when none of the previous conditions are true. It does not take a condition of its own rather it simply runs when all checks fail. THis is useful for handling default cases, unexpected inputs, or situations where you want to guarantee that at least one block of code executes.

Example:

```
temperature = 30

if temperature > 80:
    print("Hot")
elif temperature > 60:
    print("Warm")
else:
    print("Cold")
```

Note:
Because **else** has no condition, it should be used sparingly and only when a true catch-all case is needed.

Nested Conditions:

Python allows conditions to be nested inside one another. While nesting can express more complex logic, it should be used carefully to avoid deeply indented code that could become difficult to read. Often, nested conditions can be replaced with combined logical expression using **and, or,** or **not**.

Example:

```
age = 20
has_id = True

if age >= 18:
    if has_id:
        print("Entry allowed")
```

Note:
By understanding when to nest and when to combine conditions will lead to cleaner, more maintainable code.

True Condition:

Python evaluates conditions based on the concept of truthiness. Some values are inherently considered **False**:

- 0
- 0.0
- "" (empty string)
- [], {}, set() (empty collections)
- None
- False

All other values are considered **True.**

Example:
python
```
if []:
    print("This will not run")
```

Note:
Truthiness allows the user to write expressive conditions without explicitly comparing values.

Mini-Exercises to try:

1.) Write a program that checks whether a number is positive, negative, or zero.
2.) Ask the user for their age and print whether they are a child, teenager, adult, or senior.
3.) Use elif to categorize a test score into letter grades.
4.) Write a nested condition that checks if a user is logged in and has admin privileges.
5.) Demonstrate truthiness by writing conditions that test empty and non-empty values.

Loops

Python Loops

Loops allow a program to repeat actions without writing the same code multiple times. They are essential for processing collections, performing repeated calculations, and building interactive or data programs. Python provides two primary looping structures **for** loops and **while** loops each designed for different types of repetition. Understanding how loops work, how they control execution, and how they interact with conditions is a key step toward writing efficient Python programs.

The for Loop

The for loop in Python is used to iterate over a sequence of values. Unlike languages that use numeric counters by default, Python's **for** loop works directly with the element of a list, strings, tuple, or any iterable object. This design makes loops more readable and reduces the chance of errors. The loop automatically stops when it reaches the end of the sequence.

Example:

```
for letter in "python":
    print(letter)
```

Note

The loop variable(**letter** in the example above) takes on each value from the sequence one at a time. Python handles the iteration internally, so you do not need to manage indexes unless you specifically want them.

The range() Function

When you need to loop a specific number of times, Python uses the **range()** function. **range()** generates a sequence of numbers without storing them all in memory, which makes it efficient even for large loops. It can take one, two, or three arguments to control the start, stop, and step values.

Example:

```
for i in range(1, 6):
    print(i)
```

Importance:

range() gives you precise control over numeric iteration and is commonly used for counters indexing, and repeated actions.

The while Loop

The **while** loop repeats as long as its condition remains true. It is useful when the number of repetition is not known in advance and depends on the user input, sensor data, or changing program state. Because **while** loops rely on conditions, they require careful design to avoid infinite loops.

Example:

```
count = 0
while count < 3:
    print("Looping...")
    count += 1
```

Behavior Note:
The loop condition is evaluated before each iteration. If the condition is false at the start, the loop may not run at all.

Loop Control Statements

Python provides special statements that modify loop behavior:

break: Exit the Loop Early
Break immediately stops the loop, even if there are remaining iterations. It is often used when searching for a value or when a condition is met earlier than expected.

Example:

```
for num in range(10):
    if num == 5:
        break
    print(num)
```

continue: Skip to the next iteration
continue stops the current iteration and move directly to the next one. This is useful for skipping unwanted values or filtering data inside a loop.

Example:

```
for num in range(5):
    if num == 2:
        continue
    print(num)
```

Why These Matter:
Control statements give the user fine control over loop execution, making loops more flexible and expressive.

Nested Loops

Python allows loops inside other loops. Nested loops are useful for working with multi-dimensional data, generating combinations, or processing grids. However, they can become expressive in terms of performance, especially when both loops grow large.

Example:

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

Behavior Note:
The inner loop completes all its iteration for each iteration of the outer loop, which can lead to quadratic behavior.

Iterating Over Different Data Types

Python's loop system is built around the concept of iterables, meaning any object that can return its element one at a time. This includes:

- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Files
- Custom objects that implement iteration

Example:

```
for key in {"a": 1, "b": 2}:
    print(key)
```

Note:
Understanding iteration allows you to loop over almost any structure in Python, not just numeric ranges.

Mini-Exercises to try

1.) Write a for loop that prints each character of a user-entered string.
2.) Use range() to print the numbers from 1 to 20.
3.) Create a while loop that asks the user for input until they type "stop".
4.) Use continue to skip printing even numbers in a loop.
5.) Use break to stop a loop when a number greater than 50 is entered.

Lists

Python List

Lists are one of Python's most important and versatile data structures. They store ordered collections of items and allow you to add, remove and modify elements freely. Unlike arrays in lower level languages, Python lists can hold values of different types within the same structure, making them flexible for real world tasks such as data processing, user input storage, and dynamic program behavior. Understanding how lists work, especially their mutability and reference behavior, is essential before learning more advanced structures like dictionaries or classes.

Creating List

A list is created using square brackets [] with elements separated by commas. List can contain numbers, strings, booleans, other lists, or even custom objects. Python stores each element as a reference, meaning the list holds pointers to objects rather than objects themselves.

Example:
fruits = ["apple", "banana", "cherry"]
mixed = [1, "two", 3.0, True]

Note:
Because list store reference, modifying a mutable element inside a list affects the original object, not just the list entry.

Indexing and Slicing

Lists preserve the order of elements, and each element can be accessed using an index. Python uses zero based indexing, meaning the first element is at index 0. Slicing allows you to extract a portion of the list using syntax **list[start: end]**, where the end index is non-inclusive.

Example"
numbers = [10, 20, 30, 40, 50]

print(numbers[0])   # 10
print(numbers[1:4])   # [20, 30, 40]
print(numbers[-1])   # 50

Importance:
Slicing creates a new list, not a view of the original, which prevents accidental modifications.

List Mutability

Lists are mutable, meaning their contents can be changed after creation. You can modify elements, append new ones, or remove existing ones. This mutability makes a list powerful, but also introduces potential pitfalls when multiple variables reference the same object.

Example:
```
a = [1, 2, 3]
b = a
a.append(4)

print(b)   # [1, 2, 3, 4]
```

Note:
Both **a** and **b** reference the same list object. Mutating one affects the other. THis is one of the most important behaviors to understand when working with lists.

Common List Methods

Python provides many built-in methods for working with lists. These methods modify the list in place unless otherwise noted.

Example:
```
items = [3, 1, 2]

items.append(4)      # Add to end
items.insert(0, 10)  # Insert at index
items.remove(1)      # Remove first occurrence
items.sort()         # Sort list
items.reverse()      # Reverse order
```

Importance:
These methods mutate the list, they can have side effects if other variables reference the same list.

Iterating Over Lists

Lists are iterable, meaning you can loop through their elements directly. This is the preferred way to process list data in Python because it is clean, readable, and avoids index-related errors.

Example:
```
for fruit in ["apple", "banana", "cherry"]:
    print(fruit)
```

Note:
Iteration retrieves each element in order, but does not modify the list unless you explicitly change its content inside the loop.

List Comprehensions

List comprehensions provide a concise way to create a list using a single expression. They combine looping and optional filtering into a compact syntax. While powerful it should be used carefully to maintain readability.

Example:
```
squares = [x * x for x in range(5)]
```

Note:
List comprehensions often replace longer loops and make data transformations more expressive.

Nested Lists

List can contain other lists, creating multi-dimensional structures. These are useful for representing grids, tables, or matrices. However, nested lists require careful handling because modifying inner lists affect all references to them.

Example:
```
matrix = [
    [1, 2],
    [3, 4]
]
```

Behavior Note:
Nested lists introduce multiple layers of mutability, which can lead to unexpected behavior if not managed carefully.

Copying List

Since lists are mutable, copying them requires attention. Using **=** creates a reference copy, not a new list. To create an actual copy, you can use slicing, the **list()** constructor, or the **copy()** method.

Example:
```
a = [1, 2, 3]
b = a[:]       # New list
c = a          # Reference copy
```

Common Mistakes:
Shallow copies only duplicate the outer list. Nested lists still share references.

Common Pitfall to avoid for list

1.) When modifying a list while iterating over it: This can cause skipped elements or unexpected behavior.
2.) When you forget that lists are mutable: Changes propagate through all references.
3.) Using **list1 = list2** when you meant to copy: THis creates two names pointing to the same object.
4.) Confusing the slicing behavior: Slicing returns a new list not a view.

Mini-Exercises to try

1.) Create a list of five numbers and print the first, middle, and last elements.
2.) Use slicing to extract the first three elements of a list.
3.) Demonstrate list mutability by showing how two variables referencing the same list behave.
4.) Write a loop that prints each element of a list on a separate line.
5.) Use a list method to sort a list of numbers.
6.) Create a list comprehension that generates the squares of numbers from 1 to 10.

Dictionaries

Python Dictionaries

Dictionaries are one of Python's most powerful and flexible data structures. They store data as key value pairs, allowing you to access value using meaningful identifiers instead of numeric indexes. Dictionaries are optimized for fast lookups, insertions, and deletions, making them ideal for tasks involving structured data, configuration settings, user profiles, and any situation where you need to associate one piece of information with another. Understanding how dictionaries work internally, especially hashing and mutability, is essential for writing efficient and reliable Python programs.

Creating Dictionaries

A dictionary is created using curly braces {} with key value pairs separated by colons. Keys must be immutable (strings, numbers, tuples), while values can be any type. Python stores dictionary entries in a hash table, which allows extremely fast access.

Example:

```
person = {
    "name": "Shoeb",
    "age": 20,
    "student": True
}
```

Behavior Note:
Dictionaries do not store items in numeric order. Instead, they maintain insertion order, but this is a feature of the implementation, not a sorting guarantee.

Accessing and Modifying Values

Values are accessed using the keys. If a key does not exist, Python raises a **KeyError**. You can add or update entries simply by assigning a value to a key.

Example:

```
person["age"] = 67       # Update
person["city"] = "NYC"   # Add new key
```

Importance:
Dictionaries behave like dynamic objects; you can add, remove, or modify fields at any time, which makes them ideal for flexible data structures.

Dictionary Methods

Python provides the user with many built in methods for working with dictionaries. These methods allow you to retrieve keys, values, or both, and to safety access data without raising errors.

Example:

```
person.get("name")       # Safe access
person.keys()            # All keys
person.values()          # All values
person.items()           # Key-value pairs
person.pop("age")        # Remove a key
```

Note:
get() returns **None** (or a default value you specify) instead of raising an error, making it safer for uncertain keys.

Iterating Over Dictionaries

Dictionaries are iterable, but iterating directly over a dictionary returns its keys. You can also iterate over key values or key value pairs using **.value()** and **.items()**.

Example:

```
for key, value in person.items():
    print(key, value)
```

Note:
Iteration is essential for processing structured data, especially when reading from files, APIs, or user input.

Mutability and Reference Behavior

Dictionaries are **mutable**, meaning their contents can be changed after creation. Like lists, dictionaries store references to objects, not the objects themselves. THis means modifying a value inside a dictionary affects all references to that dictionary.

Example:

```
a = {"x": 1}
b = a
a["x"] = 2

print(b)   # {'x': 2}
```

Common Mistakes
If multiple variables reference the same dictionary, changes through one variable affect all others.

Keys Must Be Immutable

Dictionary keys must be immutable types because Python uses hashing to store and retrieve values. Mutables types like lists cannot be used as keys because their hash value can change, breaking the dictionary's internal structure.

Valid Keys
- strings
- integers
- floats
- tuples

Invalid Keys
- lists
- dictionaries
- sets

Example:
```
# Valid
coords = {(1, 2): "point"}

# Invalid
# { [1, 2]: "point" }  # Error
```

Note:
This rule ensures that dictionary lookups remain fast and reliable.

Nested Dictionaries

Dictionaries can contain other dictionaries, creating multiple level structures. This is useful for representing JSON data, configuration files or hierarchical information.

Example:
```
student = {
    "name": "Tai",
    "grades": {
        "math": 67,
        "science": 85
    }
}
```

Note:
Nested dictionaries allow you to model complex data without creating custom classes.

Copying Dictionaries

Like lists, dictionaries require copying. Using **=** creates a reference copy while **.copy()** or **dict()** creates a shallow copy. For nested dictionaries, used **deepcopy()**.

Example:
```
a = {"x": 1}
b = a.copy()   # New dictionary
c = a          # Reference copy
```

Note:
Shallow copies do not duplicate nested dictionaries; inner structures are still shared.

Mini Exercises to try

1.) Create a dictionary representing a book with title, author, and year.
2.) Add a new key to the dictionary and update an existing one.
3.) Loop through a dictionary and print each key and value.
4.) Use `.get()` to safely access a key that may not exist.
5.) Create a nested dictionary and access a value inside the inner dictionary.

Tuples & Sets

Python Tuples and Sets

Tuples and sets are two important built-in collection types in Python, each designed for a specific purpose. A **tuple** is an ordered, immutable sequence used for fixed collections of items, while a **set** is an unordered collection of unique elements optimized for fast membership testing and mathematical operations. Understanding how these structures behave, especially their immutability, hashing rules, and performance characteristics, helps you choose the right tool for each task.

What Is a Tuples

A tuple is an **ordered**, **immutable sequence** of values. Once created, its contents cannot be changed, added to, or removed. This immutability makes tuples ideal for representing fixe data, such as coordinates, dates, RGB values, or any group of items that should not be modified. Tuples also support indexing, slicing, and iteration just like lists, but with the added guarantee that their structure remains constant.

Example:
```
point = (10, 20)
colors = ("red", "green", "blue")
```

Note:
Because tuples are immutable, Python can store them more efficiently and use them as dictionary keys or set elements, something list cannot do.

Tuple Indexing and Slicing

Tuples preserve order, so you can access elements by index or extract sub-tuples using slicing. These operations behave exactly like list indexing, but always return new tuples.

Example:
```
nums = (1, 2, 3, 4)

print(nums[0])     # 1
print(nums[1:3])   # (2, 3)
```

Note:
Slicing a tuple never modifies the original; it always produces a new tuple.

Tuple Unpacking

Tuple unpacking allows you to assign multiple variables at once. This feature makes tuples especially useful for returning multiple values from functions or representing structured data.

Example:
x, y = (10, 20)

Note:
Unpacking works with any iterable, but tuples are the most common structure used for this pattern.

When to use Tuples Instead of Lists

Use a tuple when:

- The data should not change
- You want to ensure integrity
- You need a hashable type
- You want slightly better performance

Use a list when the data is dynamic and will be modified.

Common Mistake with Tuples

- Forgetting the comma in a single element tuple:

Example:
```
x = (5)     # Not a tuple
x = (5,)    # Tuple
```

- Assuming immutability means nested objects are protected (tuples can contain mutable items).

What Is a Set?

A set is an **unordered collection of unique elements**. Sets automatically remove duplicates and are optimized for fast membership testing. They are ideal for tasks involving uniqueness, filtering, or mathematical operations like union and intersection. Because sets are unordered, they do not support indexing or slicing.

Example:
```
numbers = {1, 2, 3, 3, 2}
print(numbers)   # {1, 2, 3}
```

Note:
Sets use hashing internally, which is why they require elements to be immutable.

Adding and Removing Elements

Sets are mutable, so you can add or remove elements after creation. However, because sets are unordered, you cannot control where elements appear.

Example:
s = {1, 2, 3}
s.add(4)
s.remove(2)

Importance:
Set operations are extremely fast, often faster than lists because they use hash tables.

Set Operations

The set supports mathematical operations that make them powerful for data analysis and filtering.

Examples:
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b) # Union: {1, 2, 3, 4, 5}
print(a & b) # Intersection: {3}
print(a - b) # Difference: {1, 2}
print(a ^ b) # Symmetric difference: {1, 2, 4, 5}

Notes:
These operations return new sets and do not modify the originals unless you can use in-place version(**|=, &=,** etc).

When to Use Sets

Use a set when:

- You need to remove duplicates
- You need fast membership checks
- You're performing mathematical set operations
- Order does not matter

Do not use a set when order or indexing is important.

Mini-Excercises

1.) Create a tuple representing a date (year, month, day) and unpack it into variables.
2.) Slice a tuple to extract the middle elements.
3.) Create a set from a list with duplicates and show the cleaned result.
4.) Use a set to check whether a user-entered value appears in a collection.
5.) Perform union and intersection on two sets of numbers.
6.) Create a tuple containing a list and modify the list—observe the behavior.

Functions

Python Function

Functions are reusable blocks of code that perform a specific task. Instead of repeating the same logic in multiple places, you define it once in a function and call it whenever you need it. This makes programs easier to read, maintain, and extend. Function also helps break large problems into smaller pieces.

Defining a Function

A function definition creates a name block of code that Python stores for later use. The **def** keyword introduces the function, followed by its name, parameters in parentheses, and a colon. Everything indented under that line belongs to the function body. Defining a function does not execute it, it only prepares it.

Example:

```
def greet():
    print("Hello!")
```

Note:
Separating definition from execution is what makes function reusable and composable.

Calling a Function

Calling a function tells Python to execute the code inside it. When a function is called, Python jumps into the function body, runs its statement, and then returns to the point where it was called. You call a function by writing its name followed by parentheses.

Example:

```
greet()  # Output: Hello!
```

Note:
If a function expects arguments and you don’t provide them, Python raises a **TypeError.**

Parameters and arguments

Parameters are placeholders defined in the function header; while arguments are the actual values you pass when calling the function. Parameters allow the same function to work with different data. Inside the function, parameters behave like local variables.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alex")
```

Importance:
Parameters make functions flexible and reusable instead of hard-coding values.

Return values

A function can send a result back to the caller using the **return** keyword. Once **return** is executed, the function stops running and the returned value can be stored or used in expressions. If no **return** is specified, the function returns **None** by default.

Example:

```
def add(a, b):
    return a + b

result = add(3, 5)  # result = 8
```

Note:
Even with multiple values, Python is returning a single tuple under the hood.

Default arguments

Default arguments let you specify fallbacks values for parameters. If the caller omits that argument the default is used. This makes the function easier to call in common cases while still allowing customization.

Example:

```
def greet(name="friend"):
    print(f"Hello, {name}!")

greet()         # Hello, friend!
greet("Alex")   # Hello, Alex!
```

Common Mistake

Default values are evaluated once at definition time using mutable defaults like lists can cause unexpected share state.

Keyword arguments

Keyword argument allows you to call a function by explicitly naming the parameters. This improves readability and lets you pass arguments in any order. It's especially helpful when functions have many parameters.

Example:

```
def introduce(name, age, city):
    print(f"{name} is {age} and lives in {city}.")

introduce(age=20, name="Alex", city="Philadelphia")
```

Importance

Keyword arguments make function calls self-documenting and reduce mistakes with parameter order.

Variable length argument

Sometimes you don't know how many arguments a function will receive. ***args** collects extra positional arguments into a tuple, and ****kwargs** collects extra keyword arguments into a dictionary. This makes functions highly flexible.

Example:

```
def total(*numbers):
    return sum(numbers)

print(total(1, 2, 3))  # 6

def show_info(**data):
    print(data)

show_info(name="Alex", age=20)
```

Behavior Note:
Use ***args** when you want any number of values and ****kwargs** when you want any number of named options.

Scope and lifetime of variables

Variables created inside a function are **local** to that function and exist only while the function runs. Variables defined outside any function are **global** within that file. If a local variable has the same name as a global one, the local version is used inside the function.

Example:

```
x = 10  # global

def func():
    x = 5  # local
    print(x)  # 5

func()
print(x)  # 10
```

Note:
Understanding scope prevents accidental overwriting and makes it clear where data is coming from.

Global and nonlocal

global allows a function to modify a variable defined at the module (file) level. **Nonlocal** allows a nested function to modify a variable from its enclosing function. Both break the usual isolation of scopes and should be used sparingly.

Example:

```
count = 0

def increment():
    global count
    count += 1

def outer():
    x = 10
    def inner():
        nonlocal x
        x = 20
    inner()
    return x
```

Common Mistake
Overusing **global** and **nonlocal** makes code harder to reason about without debugging.

Function as first class object

In Python, functions are values like any other. You can assign them to variables, store them in lists or dictionaries, pass them as arguments, and return them from other functions. This enables patterns like callbacks and higher order functions.

Example:

```
def greet():
    print("Hello")

func = greet
func()  # Hello
```

Note:
Treating functions as data is the foundation for decorators, event handlers, and many advanced techniques.

Nested Functions

You can define a function inside another function. The inner function can access variables from the outer function, forming a closure. Nested functions are useful for encapsulating helper logic that should not be visible outside.

Example:

```
def outer():
    message = "Hi"
    def inner():
        print(message)
    inner()

outer()
```

Mini-Excercise to try

1.) Write a function that returns the square of a number.
2.) Create a function with a default argument and call it with and without that argument.
3.) Write a function that returns both the minimum and maximum of a list of numbers.
4.) Use **`*args`** to create a function that multiplies any number of values.
5.) Use **`**kwargs`** to build a function that prints a formatted user profile.
6.) Demonstrate the difference between a local and a global variable with the same name.

Modules & Imports

Python Modules & Imports

Modules allow you to organize Python code into separate files so your program stays clean, reusable. Instead of writing everything in one giant script, you can split functionally into logical units and import only what you need. This makes your codebase more modular which encourages better design practices. Understanding modules is important before working with larger projects, libraries, or object oriented programming.

Module

A module is simply a Python file that contains variables, functions, or classes. When you import a module, Python loads that file and makes its contents available to your program. Modules help you avoid duplication by letting you reuse code across multiple scripts. They also make it easier to structure programs into meaningful parts.

Example:

```
# file: math_tools.py
def add(a, b):
    return a + b
```

Note:
Any Python file can become a module you don't need special syntax or configurations.

Importing a Module

The **import** keyword loads a module into your program. Once imported, you can access its contents using dot notation. Python imports a module only once per program run, storing it in memory for efficiency.

Example:

```
import math_tools

print(math_tools.add(3, 5))
```

Importing Specific Item

You can import only the parts of a module you need using the **from** keyword. This makes the code cleaner and avoids repeating the module name. However, it can also hide where a function came from if overused.

Example:
```
from math_tools import add

print(add(3, 5))
```

Note:
Avoid **from module import *** it can pollute your namespace and make debugging harder.

Importing with Aliases

Aliases let you shorten long module names or avoid naming conflicts. This is common in data science libraries like NumPy or Pandas. Aliases improve readability when used consistently.

Example:
```
import math_tools as mt

print(mt.add(3, 5))
```

Note:
Aliases help keep code concise without losing clarity.

Standard Library Modules

Python includes a large collection of built-in modules known as the **standard library**. These modules provide tools for math, date, random numbers, file handling, system operations, and more. Using the standard library saves a lot of time because you don't need to remake common functions.

Example:
```
import math
import random

print(math.sqrt(16))
print(random.randint(1, 10))
```

Behavior Note:
Standard Library modules are optimized and tested; you should always check if Python already provides what you need before writing your own.

Creating Your Modules

You can create your own modules simply by writing code in a python file. Any functions, variables, or classes defined in that file become importable. This is how you build multi-file and reusable utilities.

Example:
```
# file: greetings.py
def hello(name):
    print(f"Hello, {name}!")

# file: main.py
import greetings
greetings.hello("Shoeb")
```

Note:
Custom modules let you organize large programs into logical parts.

The Module Search Path

1. The current working directory
2. Installed packages
3. Standard library
4. System paths

This search behavior is controlled by **sys.path**, a list of directories Python checks.

Example:
```
import sys
print(sys.path)
```

Common Mistakes
If two modules share the same name, Python may import the wrong one depending on the search path.

Packages brief information

A package is a folder containing multiple modules, usually with an **__init__.py** file. Packages allow you to group related modules together, forming larger libraries. This is how major frameworks like Numypy structure.

Example structure
my_package/
  __init__.py
  tools.py
  helpers.py

Note:
Packages make it possible to build scalable, multi-file Python projects.

Mini-Exercises

1.) Create a module with two functions and import it into another file.
2.) Use from module import name to import a single function.
3.) Import a module using an alias and call one of its functions.
4.) Use the math and random modules to generate a random square root.
5.) Print the module search path using sys.path.
6.) Create a simple package with two modules and import from it.

File Handling

Python File Handling

File handling allows Python programs to read, write, and modify data stored on the computer's filesystem. This is important for working with text files, logs, configuration files, saved user data, and more. Instead of keeping everything in memory, file handling lets your program store information permanently. Understanding how to open, read write and safely close files is a core skill for building real applications,

File Opening

Python has the built in **open()** function to access files. When you open a file, Python creates a connection between your program and the file on disk. You must specify both name and the mode (read, write, append, etc). Opening a file does not automatically read or write anything it prepares the file for use.

Example:
file = open("data.txt", "r")

Note:
If the file does not exist and you open it in read more ("r"), Python raises a **FileNotFoundError.**

File Modes

File modes tell Python what you want to do with the file. The most common more the following:

- "r" read (file must exist)
- "w" write (create or overwrite file)
- "a" append(adds to end of file)
- "r+" read and write
- "b" binary mode (used for images, audios, and etc)

Example:
file = open("data.txt", "w")

Importance:
Choosing the wrong mode can erase data or prevent your program from accessing the file.

Reading from a File

Python provides several ways to read contents. You can read the entire file at once, read line by line or, read a specific number of characters. The method you choose depends on the size of the file and how you want to process it.

Example:

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

Note:
read() loads the entire file into memory to avoid this for large files.

Writing to a File

Writing allows your program to store information permanently. When using **“w”** mode, Python overwrites the file: **“a”** mode adds new content to the end. Writing does not automatically add newlines; you must include them manually.

Example:

```
file = open("log.txt", "a")
file.write("New entry\n")
file.close()
```

Common Mistakes:
Forgetting **\n** results in all text being written on one line.

Closing a file

After reading or writing, you must close the file to free system resources. Closing basically ensures that all data is properly saved and prevents file corruption. Forgetting to close files can lead to memory leaks or locked files.

Example:
```
file.close()
```

Note:
Some operating systems limit how many files can be open at once, always close files when done.

Using With

The **with** statement automatically opens and closes files for you. This is the recommended way to handle files because it prevents errors and ensures files are always closed, even if something goes wrong.

Example:
```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

Note:
When the **with** block ends, Python closes the file automatically so there is no need for **file.close()**.

Reading Files Line by Line

For large files, reading line by line is more efficient than loading everything at once. Python treats files as iterable, so you can loop through them directly.

Example:
```
with open("data.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Note:
This approach uses very little memory, even for huge files

Writing Lists or Multiple Lines

You can write multiple lines at once using **writelines()**, but you must include newline characters yourself. This method is useful when writing logs or exporting data.

Example:
```
lines = ["First line\n", "Second line\n"]

with open("output.txt", "w") as file:
    file.writelines(lines)
```

Common Mistakes:
Forgetting the **\n** results in lines being glued together.

Working with Binary Files

Binary mode (**“b”**) is required for non-text files such as images, audio, or PDFs. In binary mode, Python reads and writes raw bytes instead of characters.

Example:
```
with open("photo.jpg", "rb") as file:
    data = file.read()
```

Note:
Binary mode avoids encoding issues that occur with text files.

Mini-Exercises

1.) Create a text file and write three lines to it.
2.) Read the file back and print each line without extra newlines.
3.) Append a new line to an existing file.
4.) Use a with block to read a file safely.
5.) Write a list of strings to a file using writelines().
6.) Read a file in binary mode and print the number of bytes.

Error Handling

Python Error Handling

Error handling allows your program to deal with unexpected situations without crashing. Instead of stopping the entire program when something goes wrong, Python gives you tools to detect/catch and respond to errors. This is essential for building applications that interact with users, files, networks, or external data. Understanding how exceptions work helps you write programs that fail safely and communicate clearly.

What Are Exceptions

An exception is Python's way of signaling that something went wrong during execution. When an error occurs, Python "raises" an exception and stops the normal flow of the program. If the exception is not handled, the program crashes and prints a traceback. Exceptions allow you to separate normal logic from error handling logic making the code cleaner and more robust.

Example
```
print(10 / 0)  # ZeroDivisionError
```

Behavior Note:
Exceptions are objects; each error type is a class that represents a specific kind of problem.

The try and except Block

The **try** block lets you run code that might fail, and the **except** block lets you handle the error if it occurs. This prevents your program from crashing and allows you to respond appropriately. You can catch specific exceptions or use a general catch-all.

Example:
```
try:
    value = int("abc")
except ValueError:
    print("Invalid number!")
```

Note:
Catching only the exceptions you expect makes debugging easier and avoids hiding real problems.

Catching Multiple Exceptions

Sometimes a block of code can fail in more than one way. Python allows you to catch multiple exception types using a tuple or multiple **exception** blocks. This gives you fine grained control over how different errors are handled.

Example:

```
try:
    result = 10 / int("0")
except (ValueError, ZeroDivisionError):
    print("Something went wrong!")
```

Note:
Python checks each **except** block in order of the first match.

The else block

The **else** block runs only if no exceptions occur in the **try** block. This is helpful for code that should run only when everything succeeds, keeping the logic clean and organized.

Example:

```
try:
    num = int("5")
except ValueError:
    print("Invalid input")
else:
    print("Conversion successful:", num)
```

Importance:
else helps separate success logic from error logic.

The finally Block

The **finally** block always runs, whether an exception occurred or not. This is ideal for cleanup tasks like closing files, releasing resources, or resetting states. Even if the program crashes or returns early, **finally** still executes.

Note:
finally runs even if you use **return** inside **try** or **except.**

Raising Exceptions Manually

You can raise exceptions yourself using the **raise** keyword. This is useful when validating input, enforcing rules, or signaling that something is wrong in your program's logic.

Example:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
```

Importance:
Raising exceptions helps you catch problems early and enforce correct usage of your functions.

Custom Exceptions

You can create your own custom exception classes by inheriting from Python's built-in **Exception** class, Custom exceptions make your code more expressive and easier to debug in large applications.

Example:

```
class InvalidScoreError(Exception):
    pass
```

Behavior Note:
Custom exceptions help distinguish your program's error from Python's built in ones.

Common Built-In Exceptions

- **ValueError** wrong type of value
- **TypeError** wrong type of object
- **ZeroDivisionError** division by zero
- **FileNotFoundError** missing file
- **KeyError** missing dictionary key
- **IndexError** list index out of range

Importance:
Knowing common exceptions helps you anticipate and handle errors before they occur.

Mini-Excercise to try

1.) Write a try/except block that catches a ValueError when converting input to an integer.
2.) Create a function that raises an exception if the user enters a negative number.
3.) Use else to print a success message only when no exception occurs.
4.) Use finally to print “Done” whether or not an error happens.
5.) Catch two different exceptions in the same block.
6.) Create a custom exception class and raise it in a function.

OOP

Python Object-Oriented Programming (OOP)

Object-oriented programming is a way of structuring programs around objects instead of just function and data. An object bundles data (attributes) and behavior (methods) into a single unit, making code easier to model after real world concepts. OOP helps you build larger programs that are modular, reusable and easier to reason about. In Python, OOP is built directly into the language through classes.

Classes and Objects

A **class** is a blueprint that defines what an object should know and what it should be able to do. An **Object** (or Instance) is a concrete version of that blueprint created at runtime. You can create many objects from the same class, each with its own data but the same structure and behavior.

Example:

```
class Person:
    pass

p = Person()  # p is an object (instance) of Person
```

Importance:
Classes let you define a reusable structure, while objects are the actual thing your program works with.

Attributes and methods

Attributes are variables that belong to an object, and methods are functions defined inside a class that operate on that object. Together, they describe the state and behavior of an object. Accessing them uses dot notation.

Example:

```
class Person:
    def __init__(self, name):
        self.name = name  # attribute

    def greet(self):      # method
        print(f"Hello, I'm {self.name}")

p = Person("Alex")
p.greet()
```

Behavior Note:
Each object gets its own copy of instance attributes, but methods are shared by all instances of the class.

__init__ method and self

The **__init__** method is a special method that runs automatically when a new object is created. It initializes that object's attributes and sets up its initial state. The **self** parameter refers to the specific instance being created or used.

Example:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Importance:
self is how methods access and modify the data that belongs to a particular object.

Instance variables vs class variables

Instance variables belong to a specific object, while **class variables** are shared across all instances of the class. Class variables are defined directly inside the class body, while instance variables are usually defined in **__init__**.

Example:
```
class Person:
    species = "Human"  # class variable

    def __init__(self, name):
        self.name = name  # instance variable

p1 = Person("Alex")
p2 = Person("Sam")
```

Common Mistake:
Accidentally modifying a class variable when you meant to change an instance variable can affect all objects.

Instance methods, class methods, and static methods

Instance methods operate on a specific object and take **self** as the first parameter. **Class methods** operate on the class itself and use **@classmethod** with **cls** as the first parameter. **Static methods** are utility functions inside a class that don't depend on either **self** or **cls.**

Example:

```
class Example:
    count = 0

    def instance_method(self):
        print("Instance method")

    @classmethod
    def class_method(cls):
        print("Class method:", cls.count)

    @staticmethod
    def static_method():
        print("Static method")
```

Note:
Use instance methods for object behavior, class methods for alternative constructors or class level logic, and static methods for related helper functions.

Inheritance

Inheritance allows a class to reuse and extend the behavior of another class. The new class (child/subclass) inherits attributes and methods from the existing class(superclass/parent). This reduces duplication and lets the user build specialized versions of general concepts.

Example:

```
class Animal:
    def speak(self):
        print("Some sound")

class Dog(Animal):
    def speak(self):
        print("Woof")
```

Importance:
Inheritance lets you define common behavior once and customize it in subclasses.

Method overriding and polymorphism

A subclass can **override** a method from its parent class to change or extend its behavior. **Polymorphism** means that different objects can respond to the same method call in different ways, depending on their class.

Example:
```
animals = [Animal(), Dog()]

for a in animals:
    a.speak()  # behavior depends on the actual object type
```

Note:
Polymorphism lets you write code that works with a general interface while allowing specific behavior per class.

Encapsulation

Encapsulation is the idea of hiding internal details and exposing only what's necessary. In Python, this is done by convention using a single underscore (***name)*** *for "internal" attributes and double underscore (*__**name**) for name-mangled attributes. This discourages direct access to internal state.

Example:
```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # "private"

    def deposit(self, amount):
        self.__balance += amount
```

Importance:
Encapsulation helps protect the integrity of an object's data and keeps your code easier to maintain.

Special methods (__str__, __repr__)

Special methods (also called “dunder” methods) let your objects integrate with Python’s built-in behavior. For example, **__str__** controls how an object is printed, and **__repr__** controls its representation in the interpreter.

Example:

```
class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"Person({self.name})"

p = Person("Alex")
print(p)  # Person(Alex)
```

Note:
Implementing special methods makes your classes feel like native Python types.

Common Pitfall

- Forgetting self in method definitions
- Treating class variables like instance variables
- Overusing inheritance instead of composition
- Making everything public and tightly coupled
- Creating classes that do too many things at once

Mini-excercises

1.) Create a Person class with name and age attributes and a greet() method.
2.) Add a class variable to track how many Person objects have been created.
3.) Create an Animal base class and two subclasses (Dog, Cat) that override a speak() method.
4.) Write a class with a “private” attribute and a method to safely access it.
5.) Implement __str__ for a class so that printing an object shows useful information.
6.) Add a @classmethod that creates an object from a string like "Alex,20".

Glossary

Glossary of Python Terms

Argument
A value passed into a function when it was called. Arguments fill the placeholders defined by parameters.

Attribute
A variable stored inside an object or class. Attributes represent the object's data.

Boolean
A data type with only two values: **True** or **False**. Used in conditions and logical expressions.

Class
A blueprint for creating objects. It defines attributes and methods that all objects of that class share.

Dictionary
A collection of key value pairs. Keys must be immutable; value can be any type.

Exception
An error that occurs during program execution. Exceptions can be caught and handled using **try** and **except**.

Function
A reusable block of code that performs a specific task. Functions can accept inputs and return outputs.

Immutable
A type whose value cannot be changed after creation (**int**, **str, tuple**).

List
A mutable ordered collection of items. Lists can grow, shrink, and be modified in place.

Loop
A structure that repeats code multiple times, such as **for** and **while.**

Method
A function defined inside a class. Methods operate on objects and use **self.**

Module
A Python file containing functions, classes, or variables that can be imported into other programs.

Mutable
A type whose value can be changed after creation (**list, dict, set**)

Object
An instance of a class. Objects contain data and behavior.

Parameter
A placeholder variable in a function definition. Parameters receive values when the function is called.

Set
An unordered collection of unique values. Useful for removing duplicates and fast membership checks.

String
A sequence of characters used to present text. Strings are immutable

Tuple
An immutable, ordered collection of items. Often used for fixed data.

Future

Next Step After This Python Guide

Now that you completed a full beginner to intermediate Python guide. Here are some topics you should learn next to continue growing as a programmer.

1.) **List Comprehensions**

A way to build lists using loops and conditions. They make your code cleaner and generally more expressive.

2.) **Lambda Functions**

Small anonymous functions used for quick operations, sorting, and functional programming patterns.

3.) **Iterators & Generators**

Tools for working with large data efficiently. Generators produce values one at a time instead of storing everything in memory.

4.) **Decorators**

Functions that modify other functions. USed heavily in frameworks like Flask and Django.

5.) **Virtual Environments**

A way to isolate project dependencies so different projects don't conflict.

6.) **Build Projects**

This is the best way to grow is to build real things. You can start with:

- Calculator
- To-Do list
- Text-based game
- File organizer

7.) **Explore Python Specializations**

    - Web Development: Flask, Django
    - Data Science: NumPy, Pandas
    - Machine Learning: TensorFlow, Scikit learn
    - Automation: PyAutoGui, Selenium
    - Cybersecurity: Scapy, pwntools

8.) **Read Other People's Code**

Open source projects are one of the best teachers. Reading code helps you understand real world patterns and practice.

9.) Keep Practicing

Programming is like a skill built through repetition. The more you code the more natural it becomes.