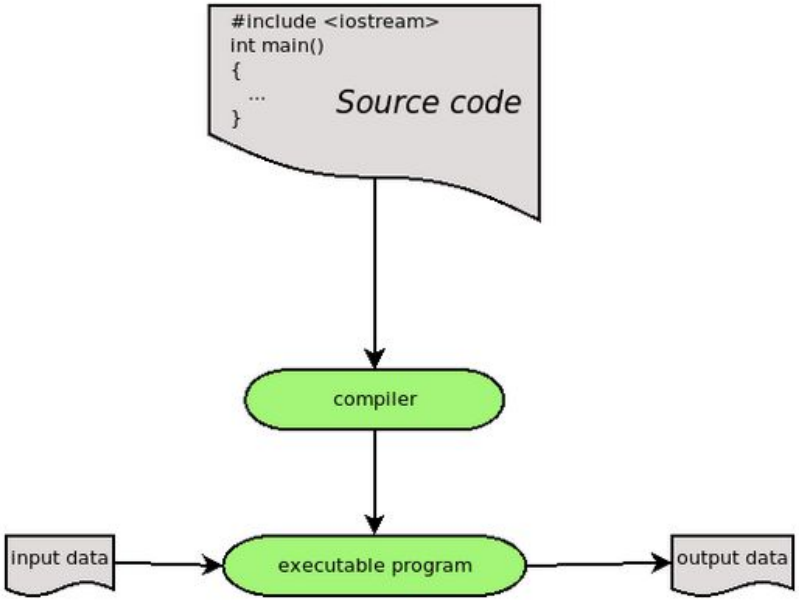
Python

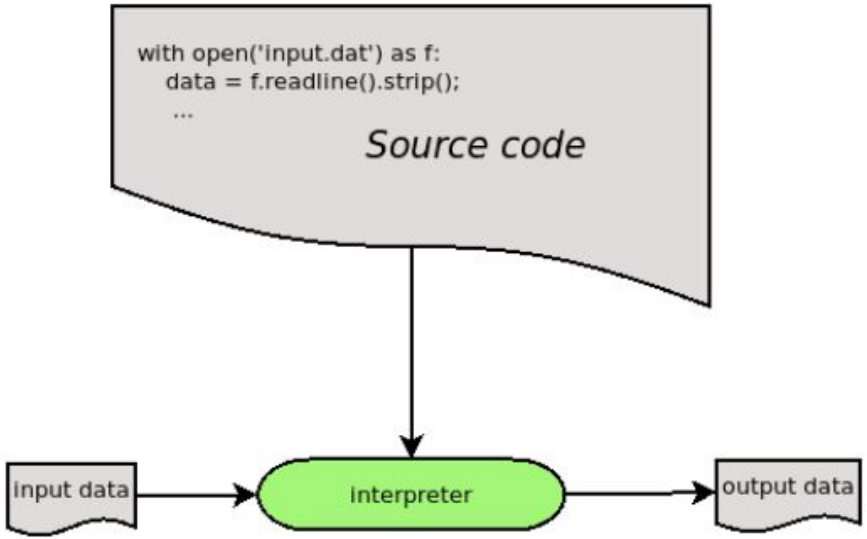
Kameswari Chebrolu

Background: Compilers and Interpreters

- Source code written as plain text in a programming language
- But computers need machine code to execute
- Two models: compilers and interpreters
- Compiler:
  - Takes program source code as input and translates into executable (binary machine code)
  - Executable runs as a process

- Interpreter:
  - takes program source code as input, reads line by line and translates internally to computation to perform
  - Simulates execution of that computation


#include <iostream>
int main()
{
...
}
Source code
compiler
input data
executable program
output data


with open('input.dat') as f:
data = f.readline().strip();
...
Source code
input data
interpreter
output data

Compiler	Interpreter
Scans the whole program in one go	Translates program line by line
Errors shown in one go at end	Errors shown line by line
Fast execution	Slow execution
Does not require source code for later execution	Requires source code for later execution
C, C++, Rust, Go	Python, Perl, Javascript

Python

- Created by Guido van Rossum, released in 1991
- Usage:
  - Web-development: django, flask, beautifulsoup, selenium
  - Data Science: numpy, pandas, matplotlib, nltk, opencv
  - ML & AI: Tensorflow, Pytorch
- Latest version Python3
  - Use IDE for heavy coding!

- Python code easy to read (closer to English)
- Python uses new lines to complete a command
  - As opposed to semicolons in other languages
- Indentation (via whitespaces) very important
  - Helps define scope (of loops, functions and classes etc)
  - As opposed to curly-brackets in other languages

Indentation

- Indentation: space at the beginning of a line of code
- Indentation in Python is very important.
  a. Other programming languages use indentation for readability
  b. Python uses indents to denote blocks of code
     - Lines of code that begin a block, end in a colon:
     - Lines within the code block are indented at the same level
     - To end a code block, remove the indentation
- Example: Below will give errors

```
if 5 > 2:

print("Five is greater than two!")

if 5 > 2:

        print("Five is greater than two!")

                print("Five is greater than two!")
```

C++ vs Python

```cpp
// Your First C++ Program

#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

```python
# This program prints Hello, world!

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

Commenting

- Comments start with a #
- Multiline?
  - Use multiple # (or)
  - Start and end with “””

print

- print() function is used to display output
  - Automatically adds a newline at the end unless specified otherwise
  - Can take multiple arguments, separated by commas
  - Automatically adds spaces between them

- Can uses f-strings (formatted string literals, print(f"...")) to create dynamically formatted output
  - Allows embedding variables, expressions, and function calls inside {} within a string
  - Supports number formatting, text alignment, and padding
  - Modern and preferred way to format strings

Variables

- No declaration needed
  - Can also change type later
- Variable names are case sensitive
  - A variable name must start with a letter or the underscore character
  - A variable name cannot start with a number
  - A variable name can only contain alpha-numeric characters and underscores
- Casting: helps specify data type
  - Can force a variable to be a specific type using int(), float(), str(), etc.

Data Types

Type	Description	Example
int	Whole numbers	`x = 10`
float	Decimal numbers	`y = 3.14`
str	Text	`name = "Alice"`
bool	True/False values	`is_active = True`

See 01-print-variables.py

Run it at terminal: python 01-print-variables.py

Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

See 02-operators.py

Strings

- Strings can be surrounded by either single quotation marks or double quotation mark
  - We will use double quotations mostly
- Strings can be indexed and sliced
  - Can be considered as arrays of characters
  - First character has index 0
  - Negative indices can be used to access characters from the end of the string

- String is an object with its own methods!
  - variety of built-in methods to perform operations like searching, replacing, changing case, trimming whitespace, and more
- Strings can be concatenated (joined together) using the + operator and repeated using the * operator.
- Strings can span multiple lines using triple quotes (''' or """)

Operator	Description	Example	Result
`+` **(Concatenation)**	Concatenates two strings.	`"hello" + " " + "world"`	`"hello world"`
`*` **(Repetition)**	Repeats a string a specified number of times.	`"hello" * 3`	`"hellohellohello"`
`in`	Checks if a substring exists within a string (returns `True` or `False`).	`"ell" in "hello"`	`True`
`not in`	Checks if a substring does not exist within a string (returns `True` or `False`).	`"bye" not in "hello"`	`True`

See 03-strings.py

Conditionals

- Allow execution of specific blocks of code based on whether a given condition is true or false
  - if statement
  - elif (else if) statement
  - else statement
- Python uses indentation to define blocks of code

Loops

- Loops are used to repeatedly execute a block of code
  - for loop
  - while loop
- Python also provides statements that can alter flow of loops
  - Break: used to exit or terminate the loop regardless of the loop's condition
  - Continue: skips the current iteration of the loop and moves to the next iteration
  - Pass: a placeholder that does nothing when executed

range()

- Used to generate a sequence of numbers
  - Often used in loops to iterate over a sequence of integers
- Syntax: range([start], stop[, step])
  - start (optional): value where the range starts
    - If omitted, the default value is 0
    - This is the first number in the sequence
  - stop (required): Value where the range ends
    - The stop value is not included in the range
    - Sequence stops just before this number
  - step (optional): value by which the sequence increments (or decrements if negative)
    - Default is 1

See 04-conditionals-loops.py

Data Structures/Collections

- List is a collection which is ordered and changeable. Allows duplicate members
  - my_list = [1, 2, 3, 4]
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members
  - my_tuple = (1, 2, 3, 4)
- Set is a collection which is unordered, changeable. No duplicate members
  - my_set = {1, 2, 3, 4}
- Dictionary is a collection which is ordered and changeable. No duplicate members
  - my_dict = {"name": "Jay", "age": 21}

- Lists: If you need to store a collection of items that may change over time (e.g. shopping cart items)
- Tuples: When you have a collection of items that will not change (e.g. pin codes)
- Set: Need to store a collection of unique elements and the order of elements doesn't matter
- Dictionary: If you need to store data as key-value pairs and require fast lookups based on keys

Data Structure	Mutable	Ordered	Allows Duplicates	Syntax Example
List	Yes	Yes	Yes	`my_list = [1, 2, 3, 4]`
Tuple	No	Yes	Yes	`my_tuple = (1, 2, 3, 4)`
Dictionary	Yes	Yes (from 3.7+)	No (keys only)	`my_dict = {"key": "value"}`
Set	Yes	No	No	`my_set = {1, 2, 3, 4}`

- Lists, sets, tuples, and dictionaries are all objects
  - Come with built-in methods

See 05-lists.py, 06-tuples.py, 07-set.py, 08-dictionary.py

Comprehension

- Concise way to create collections like lists, sets, and tuples by iterating over a sequence and optionally applying conditions or transformations
- List Comprehension: Creates a list
  - Syntax: [expression for item in iterable if condition]
    - expression: The operation or value that you want to store in the list
    - item: The variable that takes the value of each element in the iterable
    - iterable: The collection you are looping over (such as a list, range, etc.)
    - condition (optional): A filter to include only certain items from the iterable that meet the condition.

```
# Create a list of squares of even numbers
squares = [x**2 for x in range(10) if x % 2 == 0]
```

- Set Comprehension: Creates a set (removes duplicates)
  - Syntax: {expression for item in iterable if condition}
  - numbers = [1, 2, 2, 3, 4, 4, 5, 5, 6]
  - # Create a set of squares, duplicates will be removed automatically
  - unique_squares = {x**2 for x in numbers}

- Tuple Comprehension: Tuples cannot be created using a direct comprehension
  - can create a tuple from a generator expression.
  - Syntax: tuple(expression for item in iterable if condition)
  - squares_tuple = tuple(x**2 for x in range(10) if x % 2 == 0)

See 09-comprehension.py

Functions

- Three types of functions
  - Built-in Functions – pre-defined functions in Python (e.g., print(), len(), range(), type())
  - User-defined Functions – created by the user using the def keyword
  - Lambda Functions – Anonymous (one-line) functions using the lambda keyword

Built in Functions

Python has many built-in functions. A few examples

Basic Functions:

- print() – Displays output to the console
- input() – Takes user input as a string
- len() – Returns the length of an object (e.g., list, string)
- type() – Returns the type of an object

Numeric Functions

abs(x) – Returns the absolute value of x.
round(x, n) – Rounds x to n decimal places.
pow(x, y) – Returns x raised to the power y (x ** y).
sum(iterable) – Returns the sum of all elements in an iterable.
min(iterable) / max(iterable) – Returns the smallest/largest element.

Conversion Functions

int(x) – Converts x to an integer.
float(x) – Converts x to a floating-point number.
str(x) – Converts x to a string.
bool(x) – Converts x to a boolean (True or False).
list(iterable) – Converts an iterable to a list.
tuple(iterable) – Converts an iterable to a tuple.
set(iterable) – Converts an iterable to a set.
dict() – Creates a dictionary.

Other Useful Functions

sorted(iterable, key=None, reverse=False) – Returns a sorted list

range(start, stop, step) – Generates a sequence of numbers

eval(expression) – Evaluates a string as Python code

User defined functions:

- Defined using the def keyword, followed by a name and parentheses that may contain parameters
- function body is indented and contains the code to execute

Syntax:

```
def function_name(parameters):
    """Optional docstring explaining the function."""
    # Function body

    return value  # (Optional) Return a result
```

- By default, a function must be called with the correct number of arguments
  - Else will get error
- Can pass a variable number of arguments to a function using special symbols
  - *args (Non Keyword Arguments, tuple or list or sets)
  - **kwargs (Keyword Arguments, dictionary)

Lambda Functions: A small, anonymous function that can have multiple arguments but only one expression

- Uses the lambda keyword
- Syntax: lambda arguments: expression

See 10-functions.py

Modules

- Simply a Python file with a .py extension containing Python definitions and statements
  - It can define functions, classes, and variables, as well as include runnable code
  - You can think of it as a way to bundle related code together into a single file

Example: A module named math_operations.py contains

```
# math_operations.py
def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
```

- Create own module: simply save Python code in a file with the .py extension (e.g. math_operations.py)
- Can use a module by importing it using the import statement
  - There are different ways to import modules:
    - import module_name
    - from module_name import function_name
    - from module_name import * (imports everything)

Example:

```
# Importing an entire module
import math_operations
print(math_operations.add(5, 3))  # Output: 8

# Importing specific function from module
from math_operations import subtract
print(subtract(10, 4))  # Output: 6
```

- Standard Python Library: collection of modules included with Python that provide a wide range of functionalities
  - File I/O, regular expressions, networking, web services, and much more
- Examples of standard Python modules:
  - math (mathematical functions)
  - os (operating system interaction)
  - sys (system-specific parameters and functions)
  - random (random number generation)
  - datetime (date and time manipulation)

```
import math
print(math.sqrt(16))  # Output: 4.0
```

- Third-Party Modules: can install from external sources, usually through the Python Package Index (pip)
  - E.g. pip install numpy
  - After installation, you can use it like any other module
    - import numpy as np
- Packages in Python: a collection of Python modules organized in directories
  - A package must contain a special file called __init__.py, which can be empty or contain initialization code for the package.

```
mypackage/
├── __init__.py
├── module1.py
└── module2.py
```

In the module1.py:

```
def hello():
    print("Hello from module1!")
```

In the main program:

```
from mypackage import module1
module1.hello()  # Output: Hello from module1!
```

Or you can import specific functions from a module within a package:

```
from mypackage.module1 import hello
hello()  # Output: Hello from module1!
```

Point to Note

- When you import modules, you will see a __pycache__ directory
- Automatically created by Python to store compiled bytecode files (.pyc) generated by the Python interpreter
- Interpreter checks whether there's a corresponding .pyc file in the __pycache__ directory
  - If the .pyc file exists and is up to date with the source code, it will be used to speed up subsequent imports, as Python doesn't need to recompile the source code!

See 11-modules.py

Reading Files

- Reading from Files: use built-in open() function
  - allows to open a file in different modes, with the most common being:
    - 'r' (read mode) — To open a file for reading
    - 'rb' — To open a file in binary read mode
  - can read its content using methods like read(), readline(), or iterating over the file object itself
  - Best practice is to use the with statement
    - Automatically closes the file after reading, even if an error occurs during reading

Writing to Files

- open() function is again used, but in write ('w') or append ('a') mode
- Use write() to write a string or writelines() to write a list of strings
  - Does not add newlines automatically, so you have to explicitly include \n if you need new lines

stdin/stdout

- Can receive input from the user using the built-in input() function
  - Reads a line of text entered by the user and returns it as a string
  - If input should be converted (e.g., from text to integer or float), use casting functions like int() or float()
- Most common way to print data to the screen is by using the print() function
  - Sends the specified string or data to the terminal

See 12-read-write.py

Exception Handling

- try-except Blocks: Used to catch and handle exceptions
  - Code inside the try block is executed, and if an error occurs, the except block runs
  - Prevents program crashes
  - Instead of catching all errors, you can specify which exceptions to handle
    - Common exceptions include ZeroDivisionError, ValueError, FileNotFoundError, etc

- Use “finally” for Cleanup:
  - finally block always runs, whether an exception occurs or not
  - Useful for closing files, releasing resources, or performing necessary cleanup operations

See 13-exceptions.py

Class/Objects

- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects
  - Objects contain data (attributes) and behavior (methods).
  - Python supports OOP through classes and objects
- Classes: a blueprint for creating objects
  - defines attributes (data) and methods (functions that operate on the data).
  - Classes allow code reusability and modular design

- Object: an instance of a class
  - Has its own copy of the attributes and methods defined in the class
  - Key Elements:
    - class – Defines a new class
    - __init__ (Constructor) – A special method that runs when a new object is created
    - self – Represents the instance of the class and allows access to its attributes and methods

- Instance Variables: defined inside __init__() and belong to each individual object
  - Each object has its own copy of instance variables
- Class Variables: defined outside __init__() and are shared among all objects of the class
  - Changing a class variable affects all objects

- Methods: Functions inside a class that operate on objects
  - User-Defined Instance Methods: User creates within a class
    - Usually perform specific tasks related to the behavior of the object, such as modifying instance variables, performing calculations etc

- Built-in Instance Methods: predefined and are automatically available for use in any object of a particular class
  - Provide basic functionality, such as initialization, string representation, length calculation, or accessing elements etc
    - __init__: Method is automatically called when an object is created. Initializes the instance variables of the object
    - __str__: Method defines how an object should be represented as a string when printed
    - __len__: Method returns the length of an object, like the number of elements in a list
    - __getitem__: Method allows to retrieve an item from a collection like a list or dictionary using indexing

See 14-class-objects.py

References

- https://www.w3schools.com/python
- https://www.w3schools.com/python/python_examples.asp