Intro

SF1, Aug 23

Topics

- What is a programming language
- What is Python
- Basics of Python

What is a Programming Language (PL)

- **Computer speak:** The opposite of a natural or human language.
- **Characteristics:**
  - Same as natural languages: syntax and grammar.
  - Very rigid: One wrong move can lead to bugs and errors.
- **Every software** is made via a programming language.
- There are many programming languages:
  - Java
  - C
  - JavaScript
  - Assembly
  - Machine language (binary)

Types of Programming Languages

1. **Compiled Languages:**
   - Uses a compiler to output a new program in binary.
   - **INPUT:** A program in the given PL (e.g., "myprogram.java").
   - **OUTPUT:** An executable, a new program that is in binary.
   - **Examples:** Java, C, C++, C#
2. **Interpreted Languages:**
   - Reads each line one at a time and converts it into binary, sending it to the CPU.
   - **Examples:** Python, JavaScript

Questions

- **How are compiled and interpreted processes different?**
  - **Compiler:** Translates code from a high-level programming language into machine code before the program runs.
  - **Interpreter:** Translates code from a high-level programming language into machine code line-by-line as the code runs.
- **Will compiling run a program? Will interpreting run a program?**

    - Compiling produces an executable that does not run the program immediately.
    - Interpreting runs the program immediately.

- **Why would you use one over the other? What are the advantages of either?**

    - **Compiled Languages:** Easier to share code. Generally faster to run.
    - **Interpreted Languages:** Faster to develop code. Different versions (e.g., Python 3.1.7 vs. 3.1.5) will run the same binary code on the CPU.

What is Python

- **Free and Open Source Software (FOSS, FLOSS):**

    - Always free.
    - Anyone can view and contribute to the source code.
    - **Contrast with:** Closed source, proprietary software like Microsoft products.
    - Python is FOSS but also supported by a company.

- **Python Versions:**

    - Updated over the years.
    - Started with v1, v2 in the 2000s, and v3 today.
    - Differences between versions can be significant, so be cautious.

- **Applications of Python:**

    - Science
    - Data science
    - Industry applications
    - AI
    - Health studies
    - Big pros: Easier to learn, less complex syntax compared to some other languages.

- **Python as a Calculator:**

    - Can perform simple math operations.
    - Uses parentheses to enforce order of operations (PEMDAS).

Expressions and Tracing

- **Expression:** A line of code that can be simplified.

    - **Example:** `4 + 4 * 2`
        1. Evaluate `4 + 8` (one step in the evaluation)
        2. Result: `12` (final value)

- **Expression Evaluation:** An expression is evaluated one step at a time until a final value is reached.

- **Tracing:** The process of manually stepping through the evaluation of an expression to see how it is simplified step-by-step.

    - **Example:** Tracing `x = x + x` where `x = 4`

1. **Substitute:** Replace variables with their values only on the right side.
    - `x = 4 + 4`
2. **Result:** `x = 8`

Order of Operations (OOO)

To evaluate expressions correctly in Python, follow these steps:

1. **Substitution:** Replace variables with their current values.
2. **PEMDAS:** Apply the order of operations:
    - **Parentheses**
    - **Exponents**
    - **Multiplication and Division** (from left to right)
    - **Addition and Subtraction** (from left to right)
3. **Assignment:** Update the variable's value in the lookup table.

Topics

Review

- **Terms:**
    - **exp** (expression)
    - **eval** (evaluate)
    - **trace** (track)
    - **lookup table** (variable storage)
- **Python as a Calculator:** Python functions similarly to a calculator but with added support for variables.

New Concepts

- **Data:**
    - **Types:** Classification of data (e.g., integers, floats).
    - **Values:** The actual data being stored (e.g., `4`, `2.0`).
- **Type Function and Print Function:**
    - **`type()`:** Use this function to determine the data type of a value.
    - **`print()`:** Use this function to display the value.
- **Floats and Strings:**
    - **Floats:** Decimal numbers (e.g., `2.0`).
    - **Strings:** Text data (e.g., `"hello"`).

Type

All data in a computer, whether text, numbers, or images, has two fundamental elements:

- **VALUE || TYPE**

  Examples:

    - `4`: **Type:** `int`, **Value:** `4`

- `2.0`: **Type:** `float`, **Value:** `2.0`

Python supports both whole numbers (integers) and decimal numbers (floats), similar to how a calculator works.

Identifying Value and Type

To identify the type and value of data in Python:

- **Get the Type:** Use the `type()` function.
- **Get the Value:** Use the `print()` function.

```
number = 4
print(type(number))  # Output: <class 'int'>
print(number)        # Output: 4

decimal = 2.0
print(type(decimal)) # Output: <class 'float'>
print(decimal)       # Output: 2.0
```

Type Conversion in Python

Overview

Type conversion allows you to change data from one type to another. While this can be useful, it's important to be aware that some conversions might result in data loss. This section covers the basics of type conversion and provides examples of how to perform conversions using built-in Python functions.

Why Type Conversion?

Converting data types can be necessary for various operations, but it's essential to understand that:

- **Data Loss:** Some conversions may lead to loss of precision or information. For example:
  - Converting an integer (`3`) to a float (`3.0`) is straightforward and preserves the value.
  - Converting a float (`3.1`) to an integer (`3`) can lead to data loss, as the decimal part is discarded.

Built-in Functions for Type Conversion

Python provides built-in functions to convert data between types:

- `int()`: Converts a value to an integer.
- `float()`: Converts a value to a float.
- `str()`: Converts a value to a string.

Examples

Converting an Integer to a Float

- **Developer Way:**

```
x = 4
x = float(x)  # Converts int 4 to float 4.0
print(x)      # Output: 4.0
```

- **Hack Way:**

```
x = 4
x = x + 0.0  # Adding 0.0 to an integer results in a float
print(x)     # Output: 4.0
```

Casting in Python

The operation of turning one type into another is called casting. In Python, casting involves converting between different data types, such as from integers to floats or vice versa.

String Type

Python is better than a calculator in that it can operate on strings. Strings are sequences of text or characters. They "string" characters together and are represented by the `str` type in Python.

Strings

- **Strings**: Textual data
  - Examples: `"eric"`, `"sf1"`

Motivation for Types

Consider this operator: `+`

- Types inform the computer how the data should be dealt with.
  - `1 + 1` results in `2`
  - `"1" + "1"` results in `"11"` (concatenation)

Booleans

- **Booleans**: A simple type with 2 values
  - **True** or **False**
  - `bool` for short

- Comes with a range of new operations:
  - `AND`, `OR`, `NOT`, `XOR`, `XAND`, `NOR`
- Other new operations (logical operations):
  - `==` equality
  - `!=` inequality
  - Comparisons: `<`, `>`, `<=`, `>=`

Revised OOO

1. Substitution
2. PEMDAS
3. Logical operators
4. Variable assignment

Practice Traces

1.
```
x = 4
y = False
z = "hi"

a = 4 + 4 > x == y
a = 4 + 4 > 4 == False
a = 8 > 4 == False
a = True == False
```

Result: FALSE

2.
```
s = "hi"
s != s + s != True
"hi" != "hi" + "hi" != True
"hi" != "hihi" != True
True != True

"hi" != "hihi" and "hihi" != True
True and True
```

Result: FALSE and TRUE

3.
```
s = "py"
x = 3
len(s) >= 2 and (x != x or False)
len(s) >= 2 and (3 != 3 or False)
2 >= 2 and (False or False)
True and False
```

Result: FALSE

Conversion Notes and Functions in Python

Conversion

Bool to Int

- `False` converts to `0`
- `True` converts to `1`
- Any other integer value maps to `1`

Int to String

- Use the `str()` function to convert an integer to a string.

```
str(10)  # "10"
```

String to Int

- If the string represents a number, use the `int()` function to convert it.

```
int("10")  # 10
```

Functions in Python

Basic Function Definition

Functions are defined with a name, input parameters, and a body.

- **Example Definitions:**
  - `f(x) = 2x + 4`
  - `g(x) = x + x`
  - **Name:** `f` or `g`
  - **Input:** `x`
  - **Body:** The part after the equal sign

Calling a Function

To call a function, use its name followed by parentheses with arguments.

- **Example Call:**
  - `f(5)`

  Calling `f(5)` means:
  - **Variable Assignment:** `x = 5`
  - **Body:** `2 * x + 4`

More Complex Example

- **Function Definition:**
  - `h(x, y) = x + y`
- **Calling the Function:**
  - `h(1, 2)` results in `1 + 2`
  - `h(2, 1)` results in `2 + 1`

Python Syntax Examples

- **Function with Multiple Inputs:**
  - `def addition(x, y): return x + y`
- **Function with No Input:**
  - `def fun(): return 2`
- **Function with Many Inputs:**
  - `def fun1(a, b, c, d, e, f, ...): return a`
- **Function with No Return Value:**
  - `def print_greet(name): print("Hi " + name)`

Type Annotations in Python

Functions can include type annotations to specify the types of input parameters and return values.

- **Example:**
  - `def add(x: int, y: int) -> int: return x + y`
  - **Type Definition of a Function:**
    - `add: int -> int -> int`
  - **Type Definition of a Variable:**
    - `x: int`

Function Definition

- **Mathematical Functions:**
  - $f(x) = 2 \cdot x + 4$
  - $g(x) = x + x$
- **Components of a Function:**
  - **Name:** `g`
  - **Inputs:** `(x)`
  - **Body:** `x + x`
- **Python Function Definition:**
  - **Header:**

```
def addition(x, y):
```

  - **Body (indented by 4 spaces):**

```
    return x + y
```

- **Functions Without Input:**

```
def fun():
    return 2
```

- **Functions Without Return:**

```
def print_greet():
    print("hello")
```

Calling a Function

- **Example:** `f(5)`
  - Calls `f` with argument `5`
  - All arguments are positional

Python Type Annotations (Required in This Course)

- **Example:**

```
def addition(x: int, y: int) -> int:
```

Function Definition

- **Mathematical Functions**
  - `f(x) = 2 * x + 4`
  - `g(x) = x + x`
- **Function Components**
  - `g -> name`
  - `(x) -> inputs`
  - `x + x -> body`
- **Python Examples**
  - **Function with two inputs:**

```
def addition(x: int, y: int) -> int:
    return x + y
```

  - **Function with no input:**

```
def fun() -> int:
    return 2
```

  - **Function with no return value:**

```
def print_greet():
    print("hello")
```

Calling a Function

- `f(5)`
  - Calls `f` with argument `5`.
  - All arguments are positional.

Python: Type Annotations (required in this course)

- **Function with type annotations:**

```
def addition(x: int, y: int) -> int:
    return x + y
```

Parts of a Function

- **Type Signature**
  - `def addition(x: int, y: int) -> int:` **<-- header**
- **Function Definition**
  - `return x + y` **<-- body**
- **Important Notes**
  - Remember: Indentation is part of the function.
  - **Function Call**
    - `addition(1, 2)` **<-- call**
  - **Parameters:** `x`, `y`
  - **Arguments:** `(1, 2)` when `x = 1` and `y = 2`

Examples

- **Function Definition:**
  - `h(x, y) = x + y`
- **Function Call:**
  - `h(1, 2) = 1 + 2`

Tracing

- `def sub(x: int, y: int) -> int:`
- `return x - y`
- `sub(1, 2)`
- Positional arguments: map 1 to x and 2 to y.
  1. Automatic variable assignment -> `x = 1, y = 2`
  2. Expand function to its body (draw out the body) -> `x - y = 1 - 2 = -1`

Examples

- `f(3)` -> `(x = 3; 3 * x + 2)` -> `3 * 3 + 2` -> `11`
- `f(2)` -> `(x = 2; 3 * x + 2)` -> `3 * 2 + 2` -> `8`
- `f(5)` > `0 + x`
  1. (x = 5; 3 * x + 2) > `0 + x`
  2. (3 * 5 + 2) > `0 + 0`
  3. `17 > 0` **True**
- `f(0)` > `0 + x`

1. (x = 0; 3 * x + 2) > `0 + x`
2. (3 * 0 + 2) > `0 + 0`
3. `2 > 0` **True**

Python Code Flow and Control

Python executes code one line at a time, but it can handle branching and function calls to control the flow of execution.

Example: `if-else` Statement

```
x = 4
if x < 10:
  print("HELLO")
else:
  print("boo")

print("eyyy")



> prints HELLO then eyyy
> lines that get executed 0, 1, 2, 5

def add(x,y)
return x + y
add(1,2)
add(2,3)

lines executed: 1, 2, 3, 2, 4, 2, 5
```

How to Control the Flow in Python

Branching using if statements

- Allows conditional execution of code blocks.

Function Calls

- Execute a block of code defined in a function.

Key Points

- `def` keyword: Defines a function.
- `return` statement: Ends the function and sends a value back to the caller.

What Can Be Inside `return`

- **Function Inputs:**
  - Values passed to the function.
- **Logical Operators:**
  - Used to combine conditional statements (e.g., `and`, `or`, `not`).
- **Arithmetic Operations:**
  - Mathematical operations (e.g., `+`, `-`, `*`, `/`).
- **Casting (e.g., `int()`):**
  - Convert data types (e.g., `int()`, `str()`, `float()`).

Lists: Lots of Data, Structures

Lists are collections of items that help us organize data in a structured way.

Indexing

```
x = [2, 5, 7, 8]  # Example of a list with four elements

# Indexing starts at 0
print(x[2])  # Output: 7 (index 2)
print(x[-1])  # Output: 8 (last element)
print(x[-2])  # Output: 7 (second-to-last element)
```

Length of Lists

```
# len() returns the number of elements in the list
print(len(x))  # Output: 4
```

Function Example

```
# Function to calculate the class average using a list of integers
def class_avg(averages: list[int]) -> int:
    return sum(averages) // len(averages)
```

Creating and Modifying Lists

```
# The syntax for creating a list
l = [4, 5, 6, 7]  # A list with four elements

# Pulling a single value from the list
print(l[1])  # Output: 5 (value at index 1)

# Lists can also use negative indexing (reverse order)
```

```
print(l[-1])  # Output: 7 (last element)
print(l[-2])  # Output: 6 (second-to-last element)

# Updating a list element
l[0] = 99
print(l)  # Output: [99, 5, 6, 7] (first element changed)

# Adding a new element to the end of the list using append()
l.append(4)
print(l)  # Output: [99, 5, 6, 7, 4]

# Removing an element from the list using remove()
l.remove(4)
print(l)  # Output: [99, 5, 6, 7] (removes the first occurrence of 4)
```

Slicing Lists

```
# Slicing a list (getting a sublist)
# The syntax is l[start:end]
sublist = l[1:3]  # Slicing from index 1 to 2 (end index is exclusive)
print(sublist)  # Output: [5, 6]

# Omitting the start or end of the slice
print(l[:2])  # Output: [99, 5] (implicitly starts at index 0)
print(l[2:])  # Output: [6, 7] (implicitly goes to the end of the list)
```

Strings as Lists

```
# One more thing: Strings are essentially lists of characters
s = "hello"
print(s[1:4])  # Output: "ell" (substring from index 1 to 3)
```

Directory Structures 1st: ALL files live in a folder 2nd: Every folder is inside a folder 3rd: The top most folder is called the root by a slash and/or the nsmae of the dive.

c -docs -thing1.pdf -thing2.pdf -pics -2023 -bday.jpg -downloads

command cd descirption: changes the current working directory to

cd .. description: changes dir to the parent dir

ls descripiton: list all the files and folder in the current directory

cat description: prints the content of the file

Hidden files

In Unix-like operating systems, hidden files (or directories) are those whose name begins with a dot, like `.next`

When using `ls` (among other things) by itself, hidden files are not shown.

The `-a` (dash a) we added to ls is called a *switch* or a *flag* or an *option*. It modified the behaviour of ls to show *all* files (that's what the `a` in `-a` stands for.)

Each program has its own set of available flags.

- `cp SRC DST`
  - copies the entry located at SRC to DST
  - only copies a single *file* by default, not directories
  - use `-r` option to Recursively copy directories and their contents
- `rm FILE`
  - ReMoves an file. The deletion is permanent and the file cannot be recovered.
  - only removes a single *file* by default, not directories
  - use `-r` option to Recursively delete directories and their contents
- `mv SRC DST`
  - MoVes an entry from SRC to DST
  - the entry can be either a file or a directory; no `-r` is needed.
  - this is also used to rename entries
- `rmdir DIR`
  - Removes a DIRectory.
  - The directory must be empty for this to succeed.
- `mkdir DIR`
  - Creates a directory
- `touch FILE-NAME`
  - Creates a file with nothing in it

cd .. | cd .. | cd B cd ../../B cd u/B

Using Vim

Vim is a classic text editor that runs in the terminal. It's particularly useful for remote connections where only a terminal interface is available, as most servers don't have monitors connected. One of Vim's advantages is that it allows you to work efficiently without a mouse. Keep your fingers on the home row: `asdfghjkl` (left pinky on A, right pinky on L).

Vim Modes

Vim has two main modes:

1. **Insert Mode**:
   This is where you input text using the keyboard.

2. **Normal Mode**:
   This mode is for navigating and manipulating the text.

By default, Vim starts in **Insert Mode**.

- To switch from **Insert Mode** to **Normal Mode**, hit `Escape`.
- To switch from **Normal Mode** to **Insert Mode**, press `i`.

Basic Commands

- **Delete a Line**: In Normal Mode, navigate to the line and press `dd`.
- **Navigate in Normal Mode**: Use the keys `h`, `j`, `k`, `l`.
- **Highlight a Line**: Press `Shift + v` on the line.
- **Copy a Line**: Press `Y`.
- **Paste**: Press `P`.

Saving and Quitting

- **To Save a File**: Ensure you're in Normal Mode:
  - Press `:` (colon).
  - Type `w` and hit `Enter`.
- **To Quit Vim**:
  - Press `:` (colon).
  - Type `q` and hit `Enter`.

Tools for Software Development

We've discussed several tools for software development:

- **Vim**: Text editing.
- **Command Line (Git Bash)**: Running programs and managing files.
- **Git**: For version control.

Git

Git is a powerful tool that helps you:

1. Distribute files.
2. Keep track of changes.

Version control keeps track of all the changes in a file with time stamps, you can jump back to a previous version of a file easily

Very helpful for collaboration

Git is one technology (on the command line) that helps us manage changes. About 95% of software comapnies in 2024 use git

Git helps us keep track of changes but we still need to store the files somewhere. This is where companies like github, gitlab come into play. These companies *host* our code.

GIT CLONE On sites like github, gitlab, people post their code base. On the command line, we can download people's projects using git

git add a1.py git commit -m "good message" git push

Loops

So far we have talked a lot about expressions, code that is simplified to a value.

3 + 5 % 3 > ...

Statements are pieces of code that don't simpify but they do something

print("Hello") -> print to stdout return ___ -> contains an expression or value but ultimately it does something, it returns the flow of execution from where it came

if ___ -> contains an expression, but ultimately, it controls the flow of the code through branches

for loops and while loops

-another big control flow mechanism

recap: if -> branch the code function calls and returns -> jump "new" for and while loops -> repeats blocks of code

for loops: we have control how many times it runs and what code is run,

to control how many times it runs, we look at the "for __ in _____:"

to control what is run, we keep indented

for x in [1, 2, 3, 4, 5]: print("Hello") print("x is " + str(x))

x = 1 print("Hello") print("x is " + str(x)) x = 2 print("Hello") print("x is " + str(x)) x = 3 print("Hello") print("x is " + str(x)) x = 4 print("Hello") print("x is " + str(x)) x = 5 print("Hello") print("x is " + str(x))

python will "explode" the code for the for loop like this

for x in [1, 2, 3, 4, 5]: if x == 0 or x == 5: print(".." for y in [0,1,2]: if y == x: ....

range(100) = [1,2,3,4,5,6,...] range(n) list from 0 to n with jump 1 range(K,N) list from k to n incrementing by 1 range(K,N, step) list from k to n incrementing by step