Scope of the Documentation

This document is going to cover all of the topics that were covered during the presentation and/or are mentioned in the Powerpoint Presentation in the TPL group.

Topics

- **History of Python**: Understanding the origins and evolution of Python.
- **Major Computational Models**: Python uses an insane amount of models but our main focus will remain limited to the models covered during the course, for the most part. The models covered are listed below:
    - Imperative Computational Model
    - Logical Computational Model
    - Functional Computational Model
    - Object-Oriented Computational Model
- **Syntax and Semantics**: We will cover everything discussed about syntax and semantics in the course along with several points that are either worth discussing or we found interesting. Here is an outline for this section:
    - Case Sensitivity
    - Statement Endings
    - Comments
    - Indentation
    - Dynamic Typing
    - Multiple Assignment Statement
    - Conditional Statements
    - Loops
    - Functions
    - Importing Libraries
    - String Manipulation
    - Classes and Objects
    - Operators
    - Precedence
    - Associativity
    - Implicit and Explicit Type Casting
    - Algebraic Semantics
    - Axiomatic Semantics

    - Denotational Semantics
    - Operational Semantics
- **Python Script Compilation Process**: We will be discussing the how the Python Source Code is converted into object code and ultimately executed.

- **Pragmatics**: We will be discussing the real-world applicability of Python and the potential constraints that come with it. The points we will cover about Python's Pragmaticism include the following:

    - The Zen of Python
    - Readability
    - Dynamic typing
    - Interoperability
    - Modularity
    - Dedicated Interactive Environment
- **Real-World Applications**:

    - Data Science
    - Web Development
    - Game Programming
    - Automation and Scripting
- **Comparison and Potential Improvements**: We will be comparing Python with C++ and Java primarily since these languages share the same dominant computational model. We will compare several aspects of each language such as:

    - Syntax (Readability): How easy it is to read/write the code.
    - Performance (Speed): How fast the language executes.
    - Memory Consumption: How efficiently memory is managed.
    - Type System: Static vs. Dynamic typing.
    - Compilation/Execution: How the code is processed.
    - Libraries & Ecosystem: Availability and variety of libraries/frameworks.
    - Concurrency/Parallelism: Multi-threading and handling parallel tasks.
    - Platform Independence: How easily the code runs across platforms.
    - Error Handling: Mechanisms for handling exceptions.
    - Use Cases: Popular applications of each language.
- **Design Issues**: We will cover 4 of the primary design issues in Python (not implying that there aren't other issues with Python but these are the most significant). This section will include the following:

    - Performance
    - How Performance could be improved
    - GLobal Interpreter Lock(GIL) (No multi-threading)
    - How we might be able to side-step the GIL or at least improve the performance significantly (since the goal of multi-threading is to improve performance primarily)

- Dynamic Typing
- How to counter Dynamic Typing
- Memory Consumption
- Potential Solutions to the Memory Consumption Problem

Readers will have gained a holistic understanding of Python and its versatile capabilities by the end of this document (if you get to the end that is :). If you already know a bit of Python or wanna start learning than useful links will also be provided from where you can start your learning journey.

In [11]:
```
# Just some code to show the version of Python that I'm ganna be using for the code
import sys
print(f"All the code executed in this file is running on Python: \033[36m{sys.versi
```

```
All the code executed in this file is running on Python: 3.13.1 (tags/v3.13.1:067145
1, Dec  3 2024, 19:06:28) [MSC v.1942 64 bit (AMD64)] unless explicitly specified.
```

History of Python

Origins and Motivations

Python was developed by **Guido van Rossum** in 1989 at **Centrum Wiskunde & Informatica (CWI)** in the Netherlands. The language emerged from Van Rossum's desire to create a programming language that was both simple and highly readable. Inspired by the ABC programming language(PL), Python aimed to address issues of complexity while maintaining ease of use. ABC programming language was a language tha evolved from BASIC (which is another PL) primarily used to teach programming fundamentals btw.

Van Rossum's motivation stemmed from a need for a scripting language capable of handling system administration tasks while offering a **clean**, **structured**, and **engaging coding experience**. Interestingly, the name "Python" was not derived from the snake but rather from Monty Python's Flying Circus, a British comedy series, emphasizing Van Rossum's intention to make programming enjoyable (this was interesting according to GPT, not us).

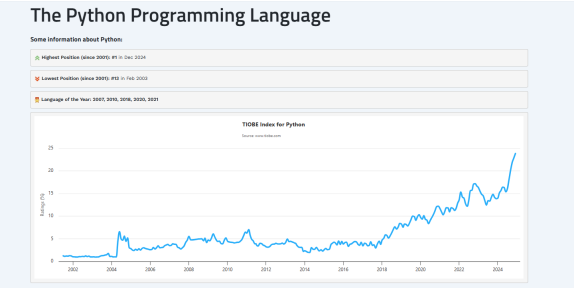
Major Milestones in Python's Evolution

- ***1989***: Python was first conceptualized as a holiday project by Van Rossum.
- ***1991***: The first public release of **Python 1.0** happened which included features such as exception handling, core data types like lists and strings, and module support. It prioritized simplicity and gave a solid foundation for future growth.

- ***1994***: Establishment of the Python Software Foundation (PSF) which was tasked with managing Python's open-source development and encouraging community contributions.
- ***2000***: Launch of Python 2.0 which introduced features such as list comprehensions, garbage collection, and improved Unicode support but it faced challenges due to backward compatibility issues, leading to divided adoption.
- ***2008***: Release of Python 3.0 which was a major overhaul aimed at addressing inconsistencies from the 2.x series (the previous versions). It introduced significant improvements such as enhanced integer division, a unified string type, and modernized input/output operations.
- ***2010s***: Python solidified its role in fields like data science and artificial intelligence. Libraries such as **NumPy**,**pandas**, and **TensorFlow** made it a lot more popular.
- ***2020s***: Continuous updates with performance optimizations. Latest stable release (as of December 23, 2024): Python 3.11, with Python 3.13 under active development, promising advancements in speed and functionality.

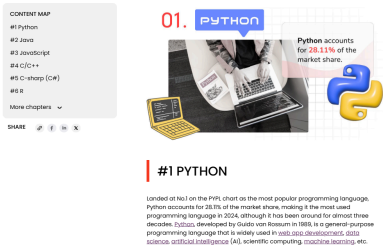
Current Day Status of Python

Python is consistently ranked as the number 1 language by popularity across the globe with perhaps the most use cases out of all the PLs.



Source: TIOBE Index

Python also has the largest market share out of all of the PLs. Although, it's worth highlighting that JS is the most popular language among Professional Developers.



Source: Orient Software

Computational Models of Python

If your basics are a bit rough then just hit this link to revise what are in Programming Languages Computational Models.

Imperative Computational Model

Imperative Programming Model is implemented into Python just like any other language. it has the same traits as usual:

- Sequential Execution.
- Variables can be reassigned throughout execution.

In [16]:
```
# here is a basic example of the imperative model in play, its just basic code for
def factorial(n):
    result = 1
    for i in range(1, n + 1):  # Sequential loop, as in sequential execution
        result *= i  # Step-by-step state mutation, as in variables are changing th
    return result

print(f"\033[31mThe factorial is: {factorial(5)}")
```

```
The factorial is: 120
```

Functional Computational Model

The Functional Model is all about **immutability**, and **pure functions**. Python supports functional programming through **higher order functions** for the most part but it also has pure functions and lambda expressions. Python may not be a purely functional language but its flexibility allows for a functional paradigm when required. Just to be clear, here is a quick summary of pure functions and higher order functions in case you guys need to revise it.

- **Pure functions have 2 primary traits**:
  - Always produces the same output for the same inputs.
  - Has no side effects (does not modify global state or interact with external systems like I/O).
- **Higher Order Functions also have 2 major traits**:
  - Takes other functions as arguments, or
  - Returns a function as its result, or
  - both of the above
- **As for Lambda expressions**, its a bit more complicated but the main traits to keep in mind are:
  - Lambda functions are anonymous (they don't have a name basically).
  - They are usually (almost always) single-line.

In [20]:
```
# Here is a simple example of Pure Functions in Python
def add(x, y):
    return x + y
# Always returns the same result for the same inputs.

a = 10
b = 20
result = add(a, b)
print(f"\033[31mThe sum of {a} and {b} is: {result}")
```

```
The sum of 10 and 20 is: 30
```

In [24]:
```
# Here is an example of Higher Order Functions in Python matee

# this function is ganna accept a function as an argument (a trait of higher order
def apply_to_each(function, data):
    result = [] # this is a list, lists are a simple collection data type in Python
    for item in data:   # oh look, that looks like a foreach loop doesn't it?
        result.append(function(item))  # append is basically adding the element int
    return result

# Defining the secondary function that we will pass in as the argument
def square(x):
    return x ** 2   ## the ** is for exponentiation, as in power, btw

# Example data
numbers = [1, 2, 3]

# Use the higher-order function
squares = apply_to_each(square, numbers)  # square is passed as an argument, which
print(f"\033[31mThe list contains the squares of the numbers that we passed(1,2,3),
```

```
The list contains the squares of the numbers that we passed(1,2,3), which are: [1,
4, 9]
```

In [9]:
```
# The following is an example of lambda expressions in python; remember, lambda exp

x = 5
double = lambda x: x ** 2        # look, it doesn't have any explicit name, and its
print(f"\033[31mSquare of {x} is: {double(x)}")  # Output: 10
```

```
Square of 5 is: 25
```

Logical Computational Model

Logical Model doesn't necessarily focus on the control flow, instead it focuses on defining relationships and logic. Python supports logical programming through various libraries (modularity) such as *pyswip* and *sympy* or even through the *prolog* library. Just keep in mind that logical programming focuses primarily on the *"what"* to solve rather than *"how"* to solve it. Logical Programming is a declarative programming paradigm btw, and declarative programming paradigm is just another way of saying that we describe the *"what"* and not the *"how"*. **Think of how SQL works**, You specify the desired result or rules; The system figures out the solution for you.

Here, some SQL code so you get the idea, its more about the *"what"* and less about the *"how"* (ik ik, mentioned it just about a thousand times, apologies for the repetition)

SELECT * FROM students WHERE grade > 90;

In [14]:
```
# We will import in some logical operators like Or and Not
from sympy import symbols, Or, Not
# But why? Doesn't Python already have the And, Or, Not operators etc? Well, yes it

# Python's and, or, not:

# Designed for immediate execution.
# Works directly with Boolean values like True or False.
# Cannot handle symbols or expressions.
# The "A AND B" assumes that both A and B are booleans and evaluates that expressio
################################################
# Logical Programming's And, Or, Not:

# Designed for symbolic computation.
# Works with symbols that don't have specific values (e.g., A, B).
# Useful for defining rules and constraints in declarative programming.
# The "A AND B" is a logical statement and A and B can be other then 0 and 1.

# Define some symbols:
Rain = symbols('Rain')
Umbrella = symbols('Umbrella')  # we are ganna try to simulate a simple "if it rain

# Now we define the logical rule.
rule = Or(Not(Rain), Umbrella)  # this just means "[(NOT rain) OR (Umbrella)]" or l

print(f"\033[31m{rule}")
```

```
Umbrella | ~Rain
```

Just so you can read the above example easily:

- | = OR
- ~ = NOT
- & = AND

In [16]:
```
from sympy import symbols, And, Or, Not

# lets run through a simple traffic light example.

# lets define our "what": "Only one light can be ON at one time"
# so we basically need 3 rules/constraints, one where only the Green light is on, o

# Define symbols like before:
Green = symbols('Green')  # symbol to represent the green light
Yellow = symbols('Yellow')  # the yellow light symbol
Red = symbols('Red')  # The red light symbol
```

```
# Logical rules:
rule = Or(And(Not(Yellow), Not(Red), Green),  # It's only green :       [(NOT Yello
          And(Not(Green), Not(Red), Yellow),  # Or it's only yellow :   [(NOT green
          And(Not(Yellow), Not(Green), Red))  # or its only red :       [(NOT Yello

print(f"\033[31m{rule}")
# Output: (Green & ~Yellow & ~Red) | (Yellow & ~Green & ~Red)
```

```
(Green & ~Red & ~Yellow) | (Red & ~Green & ~Yellow) | (Yellow & ~Green & ~Red)
```

Object Oriented Computational Model

The Dominant Model of Python

The Object-Oriented Model (OOP) is a programming paradigm that revolves around the concept of *"objects"*. These objects are instances of classes, which define a blueprint for their behavior and attributes, the same stuff as OOP that we have studied before. Python is inherently designed to support and encourage OOP, making it the dominant computational model in the language. Although its important to point out that Python heavily follows the functional paradigm too, as you can probably tell from the above examples in the Functional Programming section.

Why is Object Oriented Model the dominant model in Python?

- Everything in Python, from a simple integer dtype variable to functions is implemented in the form of objects.
- OOP is fundamental to Python's Modularity and most libraries in Python are written in OOP style (eg: tensorflow is written in an OOP style)

Python implements all the concepts of OOP whether it be **Encapsulation**, **Inheritance**, **Polymorphism**, or **Abstraction**. I am going to provide examples of each but I won't explain each of them in-depth since these concepts have been covered in previous semesters, but I'll provide additional resources that will help you learn if you so wish.

Encapsulation

In [17]:
```
class BankAccount:
    def __init__(self, balance=0):     # constructors normally have the same name a
        self.__balance = balance  # Private attribute (it will take forever if I st

# methods being defined
    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
```

```
            self.__balance -= amount
        else:
            print("Insufficient funds!")

    def get_balance(self):
        return self.__balance

# Using the class
account = BankAccount() # object created
account.deposit(1000)   # method called
account.withdraw(500)   # another method called
print(f"\033[31m{account.get_balance()}")
```

```
500
```

Inheritance

In [22]:
```
class Vehicle:
    def __init__(self, brand, wheels):
        self.brand = brand
        self.wheels = wheels

    def display_info(self):
        print(f"\033[31mBrand: {self.brand}, Wheels: {self.wheels}")

class Car(Vehicle):  # Inheriting from Vehicle
    def __init__(self, brand, wheels, fuel_type):
        super().__init__(brand, wheels)  # Calling the parent constructor
        self.fuel_type = fuel_type

    def display_info(self):  # Method overriding
        super().display_info()
        print(f"Fuel Type: {self.fuel_type}")

# Using the classes
my_car = Car("Toyota", 4, "Petrol")
my_car.display_info()
```

```
Brand: Toyota, Wheels: 4
Fuel Type: Petrol
```

Polymorphism

In [26]:
```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow Meow!"
```

```
animals = [Dog(), Cat()]
for animal in animals:
    print(f"\033[31m{animal.speak()}")
```

```
Woof Woof!
Meow Meow!
```

Abstraction

In [27]:
```
from abc import ABC, abstractmethod

class Shape(ABC):  # Abstract Base Class
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# Using the classes
rect = Rectangle(5, 3)
print("\033[31mArea:", rect.area())
print("\033[31mPerimeter:", rect.perimeter())
```

```
Area: 15
Perimeter: 16
```

Advanced OOP Concepts

Python implements almost all advanced OOP concepts as well, I will provide resources if you wanna learn about these. Here is the list of advanced OOP Concepts that are implemented in Python:

- Multiple Inheritance
- Interfaces
- Generics
- Operator/Method Overloading
- Method Overriding
- Composition
- Metaclasses

Syntax of Python

The points regarding syntax are as follows:

1. Case Sensitivity
2. Statement Endings
3. Comments
4. Indentation
5. Dynamic Typing
6. Multiple Assignment Statement
7. Conditional Statements
8. Loops
9. Functions
10. Importing Libraries
11. String Manipulation
12. Classes and Objects
13. Operators
14. Precedence
15. Associativity
16. Implicit and Explicit Type Casting

We will try to just cover all of these in code since thats much easier to explain, will also provide some theory where applicable. Lets try to make a tiny, and simple project like we used to make in our OOP Labs.

Here is what we will make:

Fantasy RPG (Role-Playing Game) Inventory and Battle System We will cover the following functionalities:

- There will be two classes: Player and Enemy.
- Player will have attributes like health, name, and also an inventory. Player will start off with a default inventory.
- Enemy will also have a name and health attribute.
- Define a function where the Player and the Enemy will fight and their healths will be dynamic, maybe let the player heal somehow too.

That will cover most of the basic syntax of Python.

In [2]:
```
import random as rd     # this is how we import libraries in Python (10 and 2✅)
# you can see we don't need to use semi-colon or anything as a statement terminator

class Player:           # this is a class in Python, and you can also see indentati
    """This is a class that is going to represent the players in our RPG game.
    Oh, and this is a multi-line comment btw, you just enclose multiple lines into
    def __init__(self, name):       # this is a constructor, which is a method in a
        self.name = name
        self.health = 120
        self.inventory = {"Sword": 1, "Battle Axe": 1, "Potion": 2} # default inven
    def drink_potion(self):
        if (self.health < 200) and (self.inventory.get("Potion", 0) > 0):
            self.health += 20
            self.inventory["Potion"] -= 1
            if self.health > 200:
                self.health = 200
            print(f"The mighty {self.name} just used a potion, their health has bee
        else:
            print(f"My apologies Dear Knight, it seems as if you have ran out of po
    def show_inventory(self):
        print("\nThis is your inventory good sir:")
        for item, count in self.inventory.items():      # A for loop applied on the
            print(f"{item}: {count}")           # (8✅)

class Enemy:
        def __init__(self, name, health, attack):
            self.name = name
            self.health = health
            self.attack = attack

def fight(player, enemy):       # (9✅)
    print(f"A frightening {enemy.name} has appeared out of the blue")
    while player.health > 0 and enemy.health > 0:           # while loop in action
        print(f"\033[32m{player.name}'s health is {player.health}.")
        print(f"\033[31m{enemy.name} has a health of {enemy.health}.")

        action = input("Choose an action {player.name} (attack, heal, run)").strip(
        if action == "attack":  # (7✅)
            damage = rd.randint(10, 30)  # Dynamic typing
            enemy.health -= damage
            print(f"\033[32m{player.name} attacked {enemy.name} for {damage} damage
        elif action == "heal":
            player.drink_potion()
        elif action == "run":
            print(f"\033[35m{player.name} fled the battle! What a fricking coward a
            return
        else:
            print("Invalid action. Try again.")
            continue

        # Enemy's turn
        if enemy.health > 0:
            damage = rd.randint(5, 45)
            player.health -= damage
            print(f"\033[31m{enemy.name} attacked {player.name} for {damage} damage
```

```
    if player.health <= 0:
        print(f"\n\033[31m{player.name} has been defeated by {enemy.name}...")
    elif enemy.health <= 0:
        print(f"\n\033[32m{enemy.name} has been defeated! Victoryyyy!")


def main():
    print("\033[34mWelcome to the Fantasy RPG!\033[0m")

    # Multiple assignment and also dynamic typing (5 and 6 ✅)
    player_name, enemy_name = "Galileo", "Friedrich Nietzsche"
    player = Player(player_name)
    enemy = Enemy(enemy_name, health=100, attack=10)

    # Swapping inventory items as an example (6 ✅)
    player.inventory["Bow"], player.inventory["Sword"] = 1, player.inventory.pop("S

    # Show initial inventory
    player.show_inventory()

    # Start battle
    fight(player, enemy)

# Run the program
if __name__ == "__main__":
    main()
```

```
Welcome to the Fantasy RPG!

This is your inventory good sir:
Battle Axe: 1
Potion: 2
Bow: 1
Sword: 1
A frightening Friedrich Nietzsche has appeared out of the blue
Galileo's health is 120.
Friedrich Nietzsche has a health of 100.
Galileo attacked Friedrich Nietzsche for 30 damage!!!
Friedrich Nietzsche attacked Galileo for 41 damage!
Galileo's health is 79.
Friedrich Nietzsche has a health of 70.
Galileo attacked Friedrich Nietzsche for 19 damage!!!
Friedrich Nietzsche attacked Galileo for 9 damage!
Galileo's health is 70.
Friedrich Nietzsche has a health of 51.
Galileo fled the battle! What a fricking coward am I right?
```

Lets quickly cover the few points that the above example didn't cover now

In [1]:
```
### Operators in Python
# Arithmetic Operators
x = 10
y = 3
print("\033[34mArithmetic Operators\033[0m")
print("\033[31mAddition:", x + y)       # 13
print("Subtraction:", x - y)            # 7
print("Multiplication:", x * y)         # 30
print("Division:", x / y)               # 3.3333
print("Floor Division:", x // y)        # 3
print("Modulus:", x % y)                # 1
print("Exponentiation:", x ** y)        # 1000
print("\033[32m==========================================================================
# Comparison Operators
print("\033[34mComparison Operators\033[0m")
print("\033[31mEqual:", x == y)             # False
print("Not Equal:", x != y)                 # True
print("Greater than:", x > y)               # True
print("Less than:", x < y)                  # False
print("Greater or Equal:", x >= y)          # True
print("Less or Equal:", x <= y)             # False
print("\033[32m==========================================================================

# Logical Operators
print("\033[34mLogical Operators\033[0m")
a = True
b = False
print("\033[31mAND:", a and b)          # False
print("OR:", a or b)                    # True
print("NOT:", not a)                    # False
print("\033[32m==========================================================================

# Assignment Operators
print("\033[34mAssignment Operators\033[0m")
z = 5
z += 2                                  # z = z + 2
print("\033[31mz after +=:", z)         # 7
z *= 3                                  # z = z * 3
print("z after *=:", z)                 # 21
print("\033[32m==========================================================================

# Membership Operators
print("\033[34mMembership Operators\033[0m")
my_list = [1, 2, 3, 4]
print("\033[31m2 in list:", 2 in my_list)   # True
print("5 not in list:", 5 not in my_list)   # True
print("\033[32m==========================================================================

# Identity Operators
print("\033[34mIdentity Operators\033[0m")
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print("\033[31ma is b:", a is b)          # True
print("a is c:", a is c)          # False
```

```
print("a == c:", a == c)          # True
print("\033[32m========================================================================

### Operator Precedence
print("\033[34mOperator Precedence\033[0m")
# Example of precedence
result = 2 + 3 * 5 ** 2 - 4 / 2
# Exponentiation first, then multiplication, division, addition, subtraction
print("\033[31mPrecedence result:", result) # 75.0
print("\033[32m========================================================================

### Associativity
print("\033[34mAssociativity\033[0m")

# Left-to-right example
print("\033[31mLeft-to-right:", 5 - 3 - 1)  # 1
# Right-to-left example (exponentiation)
print("Right-to-left:", 2 ** 3 ** 2) # 512
print("\033[32m========================================================================

### Implicit and Explicit Type Casting
# Implicit Type Casting
print("\033[34mType Casting\033[0m")

x = 5  # Integer
y = 2.5  # Float
z = x + y  # Python converts x to float implicitly
print("\033[31mImplicit Casting:", z, type(z))  # 7.5, float

# Explicit Type Casting
x = 10.7
y = "42"
z = int(x) + int(y)  # Explicit conversion to int
print("\033[31mExplicit Casting:", z)  # 52
print("\033[32m========================================================================

### String Manipulation
print("\033[34mBasic String Manipulation\033[0m")
# Basic Operations
my_str = "This is taking foreverrrr!"
print("\033[31mUppercase:", my_str.upper())
print("Lowercase:", my_str.lower())
print("Replace:", my_str.replace("foreverrrr", "MOST CERTAINLY TAKING FOREVERRRRRRR
print("Slice:", my_str[7:13])

# F-Strings
name = "David Corenswet"
age = 31
print(f"Superman's new name is {name} and he is {age} years old.")

# String Concatenation
first = "Good Morning!"
second = "How are you doing today Lois?"
print("Concatenation:", first + " " + second)
```

```
Arithmetic Operators
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Exponentiation: 1000
================================================================================
Comparison Operators
Equal: False
Not Equal: True
Greater than: True
Less than: False
Greater or Equal: True
Less or Equal: False
================================================================================
Logical Operators
AND: False
OR: True
NOT: False
================================================================================
Assignment Operators
z after +=: 7
z after *=: 21
================================================================================
Membership Operators
2 in list: True
5 not in list: True
================================================================================
Identity Operators
a is b: True
a is c: False
a == c: True
================================================================================
Operator Precedence
Precedence result: 75.0
================================================================================
Associativity
Left-to-right: 1
Right-to-left: 512
================================================================================
Type Casting
Implicit Casting: 7.5 <class 'float'>
Explicit Casting: 52
================================================================================
Basic String Manipulation
Uppercase: THIS IS TAKING FOREVERRRR!
Lowercase: this is taking foreverrrr!
Replace: This is taking MOST CERTAINLY TAKING FOREVERRRRRRRRRRRR!
Slice:  takin
Superman's new name is David Corenswet and he is 31 years old.
Concatenation: Good Morning! How are you doing today Lois?
```

Hopefully you understood most of the basic syntax of Python from the above example. Here is a quick summary of each point:

1. Case Sensitivity: Var != var, Python differs between upper and lower case.
2. Statement Endings: The statement terminator is simply a new-line instead of a semi-colon.
3. Comments: Uses # for single-line comment and a docstring ("""comment here""") for multi-line comments.
4. Indentation: Scope of the blocks of codes is defined using indentation instead of braces.
5. Dynamic Typing: Variable dtype doesn't need to be specified and it can be changed on run-time (without any type casting).
6. Multiple Assignment Statement: You can initialize as well as swap multiple variables on a single line.
7. Conditional Statements: Simple if cases and elif keyword is used for else if. Indentation is used here as well
8. Loops: Both for and while loop syntaxes are given above in the example so just check from there.
9. Functions: def keyword is used to define a function.
10. Importing Libraries: import keyword is used to import in any libraries.
11. String Manipulation
12. Classes and Objects: Syntax for classes and objects is provided both in the above example and also in the OOP part of the Computational Models section.
13. Operators
14. Precedence
15. Associativity
16. Implicit and Explicit Type Casting

Semantics of Python

Semantics define the *meaning* of the code just like syntax defines the *structure* and pragmatics defines the *implementation* of that code. Semantics ensures that programs behave as expected during execution. It is about understanding how the language interprets expressions, functions, and structures etc. We covered 4 types of semantics in the course so we will cover all 4 of them through a more Pythonic lens.

Algebraic Semantics

Algebraic semantics describes the meaning of code using mathematical structures like algebraic equations. Python's semantics can often be explained through operations defined on data types.

```
x = 5
y = 3
```

```
z = x + y  # Addition as defined mathematically
print(z)  # Output: 8
```

Python adheres to well-defined rules for operations based on the type of data, ensuring that the output aligns with expected algebraic properties.

Axiomatic Semantics

Axiomatic semantics expresses program properties using logical assertions *(preconditions and postconditions)* to guarantee correctness. In Python, this concept can be integrated using commands like *assert*.

```
x = 10
y = 5
assert x > y, "x should be greater than y"
print("Assertion passed!")
```

Axiomatic semantics ensures logical soundness and is commonly used in Python for debugging and validating conditions.

Denotational Semantics

Denotational semantics translates Python code into mathematical functions or mappings that define program behavior. This is often used to describe functions and mappings between inputs and outputs.

```
def square(x):
    return x ** 2

# Mapping: f(x) = x^2
result = square(4)
print(result)  # Output: 16
```

Python functions and constructs often follow mathematical mappings, making it easier to reason about the behavior.

Operational Semantics

Operational semantics focuses on the step-by-step execution of Python code, emphasizing how instructions are carried out by the Python interpreter.

```
x = 0
while x < 5:
    print(x)
    x += 1  # Increment
# Output: 0, 1, 2, 3, 4
```

Python's dynamic execution model interprets and executes instructions sequentially, making operational semantics essential for debugging and tracing.

Summary of Semantics

Just think that each type of semantics highlights a unique perspective:

- Algebraic: Mathematical rules for operations.
- Axiomatic: Logical correctness and preconditions.
- Denotational: Mapping inputs to outputs.
- Operational: Step-by-step execution.

The Compilation Process of a Python Script

One of the most common misconceptions about Python is that it is purely an *interpreted language*. While Python uses an interpreter, it also involves a compilation step. This is why the heading above mentions "Compilation" — Python scripts undergo a compilation phase before execution. Let's explore the backend process of how Python scripts are compiled and executed.

Step-by-Step Compilation Process

1. **Source Code (Written by Developer)**:
    - Python programs start as source code written by developers.
    - The source code consists of human-readable instructions written in `.py` files.
2. **Lexical Analysis**:
    - The first step in the backend process is **lexical analysis**, where the source code is broken into smaller components called **tokens** by the scanner.
    - Tokens include keywords, identifiers, operators, and symbols. This phase ensures that the structure of the code adheres to Python's syntax rules.
3. **Parsing (Abstract Syntax Tree - AST)**:
    - The tokens are then passed to the **parser**, which creates an **Abstract Syntax Tree (AST)**.
    - The AST is a tree representation of the structure of the source code. It organizes the code into a hierarchical format for easier processing.
    - This step ensures that the logical structure of the code is correct.
4. **Bytecode Compilation**:
    - After parsing, the AST is converted into **bytecode** — a low-level representation of the code.
    - Bytecode is platform-independent and consists of instructions that can be understood by the Python Virtual Machine (PVM).
    - Python saves the compiled bytecode into `.pyc` files in the `__pycache__` directory.
5. **Python Virtual Machine (PVM) Execution**:

- The **PVM** is the heart of Python's interpreter.
- The bytecode is fed to the PVM, which executes it step-by-step. This involves translating the bytecode into machine instructions that the underlying hardware can execute.

6. **Result**:

   - Finally, the PVM executes the code and produces the desired output or results.

Points of Confusion

1. **Why Compilation in an "Interpreted" Language?**

   - The compilation process in Python (to bytecode) improves execution speed because the source code doesn't need to be re-analyzed every time it is run.
   - The presence of `.pyc` files in the `__pycache__` directory is evidence of this compilation step.

2. **Difference Between Compilation and Interpretation in Python specifically**:

   - Compilation: Converts high-level code into an intermediate format (bytecode).
   - Interpretation: Executes the bytecode line-by-line, making Python a hybrid language.

3. **Why Is Bytecode Important? Can't we just skip the conversion to bytecode step?**

   - Bytecode allows Python to remain platform-independent.
   - By using bytecode, Python avoids repetitive parsing and reduces execution time for subsequent runs of the same script.
   - There are ways to bypass the bytecode step, it would come with several tradeoffs but it is possible. Its a much bigger topic so I will simply provide a link to some articles if you wanna explore it further.

4. **Are Scanner and Parser language specific?**:

   - Yes, every programming language typically implements its own scanner and parser because:
     - Syntax: Each language has unique syntax and rules, so the scanner and parser are designed accordingly.
     - Efficiency: They are optimized for that specific language's grammar.

5. **Is the PVM related to the virtualization and virtual machines that are discussed in Operating Systems?**

   - Nope, the PVM is not related to VMware or WSL2. PVM is a *language runtime environment* that translates bytecode into machine instructions. It has nothing to do with creation of virtual instances of an entire OS.

6. **If Python has its own VM, does that mean all other languages also have their own VMs?**

   - Well, yes and no. Some languages do have their own VMs while others don't. A VM here is like a helper that usually deals with the bytecode. Lets quickly go over a few languages
     - Java: Yes, Java has the Java Virtual Machine (JVM), which executes Java bytecode.

- C#: C# uses the Common Language Runtime (CLR), part of .NET and is used for languages like C# and F#.
- C/C++: No, these two typically compile directly to machine code since they are already quite *low-level* and do not use a virtual machine and therefore have no intermediate bytecode conversion step (just one reason why C and C++ beat most other languages when it comes to performance.).

7. Is **bytecode the same as assembly language?**

- Nope, bytecode is at a higher-level as compared to assembly language.
- Bytecode is specifically for the VMs to use.
- Bytecode is abstract and need to be translated into machine instructions through the VM.
- Think of it like this:
  - Machine Code: The language your CPU speaks directly.
  - Assembly Language: A human-readable form of machine code (e.g., MOV R1, 10).
  - Bytecode: A universal language understood by the virtual machine (e.g., LOAD_CONST, BINARY_ADD).

Step-by-Step Example

1. **Source Code**

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

2. **Lexical Analysis (Scanner)**

Token(Type: IDENTIFIER, Value: 'x')

Token(Type: ASSIGNMENT, Value: '=')

Token(Type: INTEGER, Value: '10') ... Token(Type: PRINT, Value: 'print')

Token(Type: LEFT_PAREN, Value: '(')

Token(Type: IDENTIFIER, Value: 'z')

Token(Type: RIGHT_PAREN, Value: ')')

3. **Abstract Syntax Tree (Parser)**

In [13]:
```
# No need to try to understand the code, its GPT generated, just to show you an exa
import ast
from anytree import Node, RenderTree

# Python source code
code = """
x = 10
```

```
y = 20
z = x + y
print(z)
"""

# Parse the code into an AST
parsed_ast = ast.parse(code)

# Function to recursively build the tree
def build_tree(node, parent=None):
    node_name = type(node).__name__
    tree_node = Node(node_name, parent=parent)
    for child in ast.iter_child_nodes(node):
        build_tree(child, tree_node)
    return tree_node

# Build the AST tree
root = build_tree(parsed_ast)

# Print the tree
for pre, fill, node in RenderTree(root):
    print(f"{pre}{node.name}")
```

```
Module
├── Assign
│   ├── Name
│   │   └── Store
│   └── Constant
├── Assign
│   ├── Name
│   │   └── Store
│   └── Constant
├── Assign
│   ├── Name
│   │   └── Store
│   └── BinOp
│       ├── Name
│       │   └── Load
│       ├── Add
│       └── Name
│           └── Load
└── Expr
    └── Call
        ├── Name
        │   └── Load
        └── Name
            └── Load
```

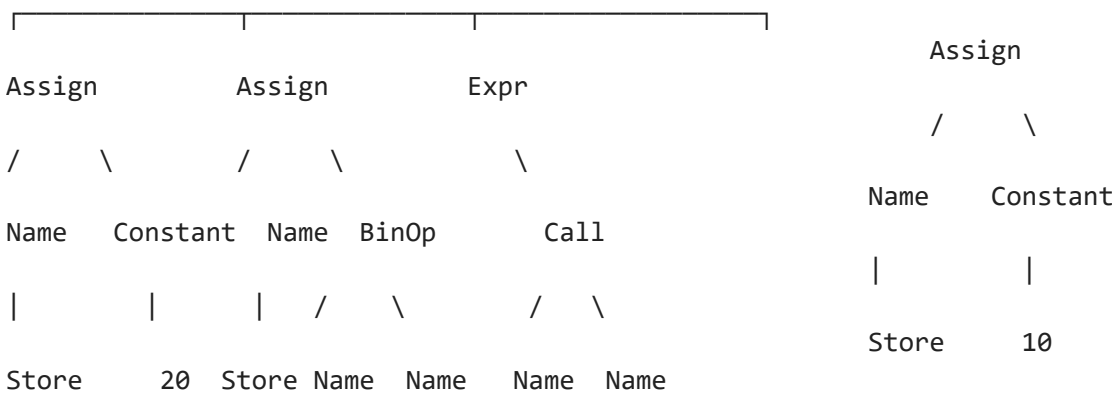
This tree is ganna be really hard to read so I'll try to explain it level by level.

1. level 1 (top):
    - This is the root of the tree, representing the entire Python program.
2. level 2:

- The root has 4 children, each representing one line of the code in this example (3 assignment statements and one expr node representing the line that has the print function)

3. Explaining Assignment Nodes:
   - These are the assignment operators. Each Assign node represents an assignment operation. We did 3 total assignments in the code so there should be 3 Assign children in the tree. Let's break them down:
     - x = 10:
       - Name: The variable x is being assigned a value.
       - Store: Indicates that x is being stored (assigned a value).
       - Constant: Represents the value being assigned (10).
     - y = 20:
       - Name: The variable y is being assigned a value.
       - Store: Indicates that y is being stored (assigned a value).
       - Constant: Represents the value being assigned 20
     - z = x + y:
       - Name: The variable z is being assigned a value.
       - Store: Indicates z is being stored.
       - BinOp: Represents the binary operation x + y.
       - Name: Refers to x and y.
       - Load: Indicates these variables are being loaded (used).
       - Add: Represents the addition operation.
4. Explaining Expr Node:
   - These are the expressions, expressions as in any statement in Python that produces a value.
     - Call: Represents a function call (in this case, print).
     - Name: Refers to the print function.
     - Load: Indicates print is being called/loaded.
     - Name: Refers to the variable z.
     - Load: Indicates z is being loaded for the function.

It might be that it is still kinda hard to visualize the tree in your head since that above output is not what we are used to. Here is me trying to draw the tree, maybe it will help visualize



```
|      |  Add  |   (print) (z)

Load     x       y
```

4. **AST to Bytecode**

In [9]:
```
import dis

def example():
    x = 10
    y = 20
    z = x + y
    print(z)

dis.dis(example)
```

```
3           RESUME                   0

4           LOAD_CONST               1 (10)
            STORE_FAST               0 (x)

5           LOAD_CONST               2 (20)
            STORE_FAST               1 (y)

6           LOAD_FAST_LOAD_FAST      1 (x, y)
            BINARY_OP                0 (+)
            STORE_FAST               2 (z)

7           LOAD_GLOBAL              1 (print + NULL)
            LOAD_FAST                2 (z)
            CALL                     1
            POP_TOP
            RETURN_CONST             0 (None)
```

The Python code is broken down into bytecode (which is platform independent). The code you see above like *RESUME* or *LOAD_FAST* or *STORE_FAST* are called opcode, as in operation code. Think of opcodes as "instructions" in a recipe. Each opcode tells the PVM what step to perform next, like "add two numbers" or "store this value in memory."

5. **From here the PVM will take over and execute the bytecode**
   - LOAD_CONST 0 (10): Load the constant 10 onto the stack.
   - STORE_NAME 0 (x): Store the value 10 in the variable x.
   - The process continues for all instructions until the final result is produced.

The Pragmatics of Python

The Zen of Python

Pragmatics focuses on the real-world usage of the language, the design trade-offs, and how the language is used in practice. Its about the implementation of the language whereas syntax was about the structure, and semantics was about the meaning of the language.

Python's guiding philosophy, "The Zen of Python," is a collection of principles that aim to make the language pragmatic, clean, and intuitive. It was written by Tim Peters who was one of the major contributors to the original implementation of Python, as in CPython. Access the "Zen of Python" by importing `this` in Python.

In [1]:
```
# Importing The Zen of Python
import this
```

```
The Zen of Python, by Tim Peters

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

Key principles include:

- **Explicit is better than implicit**: Python aims for clarity.
- **Readability counts**: Python’s syntax is intentionally designed to be readable.

- **There should be one-- and preferably only one --obvious way to do it**: This minimizes confusion for developers.

Although, Python sometimes violates its own principles. For example, there are multiple ways to achieve the same result (e.g., `map()` vs. list comprehensions).

Readability

Python is known for its clear and concise syntax, prioritizing **readability**. You have seen plenty of Python code above but here is a tiny example, and we will cover this further in the Comparison section:

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

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

Dynamic Typing

Python's **dynamic typing** allows variables to change types on the fly. The type of the variable is assigned on runtime:

```
# Dynamic Typing Example
x = 42          # x is an integer
print(type(x)) # Output: <class 'int'>

x = "hello"     # x is now a string
print(type(x)) # Output: <class 'str'>
```

- **Strengths**: Speeds up development and allows flexibility.
- **Challenges**:
  - Errors are caught only at runtime.
  - Ambiguity in large codebases.
  - We will discuss potential counters to this challenge in the design issues section.

Interoperability

Interoperability in programming languages is the ability of multiple programming languages to work together in the same system and exchange data and objects. It's an important skill for developers who work with multiple languages. For interoperability, we need a *glue languages* that enable seamless integration of multiple languages. Python serves as a great *glue language* because of its easy-to-use syntax, extensive libraries, and ability to seamlessly integrate with code written in other languages, making it ideal for connecting different components of a system together

Here are some examples:

1. **Calling C Libraries**:

```
import ctypes

# Load a C library (e.g., libc on Unix-based systems)
libc = ctypes.CDLL("libc.so.6")

# Call the printf function
libc.printf(b"Hello from C!\n")
```

2. **Using Java Libraries via Jython** (example):

```
# Sample (for Jython)
from java.util import Date

date = Date()
print(date)
```

Modularity

Python's **modular design** allows developers to build reusable and maintainable code. Python offers more then 200 modules as well as the functionality to build user-defined modules. Keep in mind that the word modularity doesn't just refer to modules/libraries only, it also refers to constructs like functions etc. Although, our focus here is on the modules/libraries that Python offers:

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

# Custom Module Example
# File: mymodule.py
def greet(name):
    return f"Oyi, its, {name}!"

# Main File
from mymodule import greet
print(greet("Agent 47, RUN!!"))  # Output: Oyi, its, Agent 47, RUN!!
```

Lets quickly cover a few of Python's popular libraries:

- **NumPy**: Provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions.
- **Pandas**: Offers powerful data structures (pandas dataframe and pandas series) for data analysis and manipulation.
- **Matplotlib**: A 2D plotting library for creating static, interactive, and animated visualizations.
- **Scikit-learn**: A machine learning library offering tools for classification, regression, and clustering.
- **TensorFlow**: An end-to-end open-source platform for machine learning.
- **Flask**: A lightweight web application framework.

- **Django**: A high-level framework for building robust web applications quickly.
- **Beautiful Soup**: A library for scraping data from HTML and XML files.
- **Requests**: A simple HTTP library for making web requests.
- **PyTorch**: A machine learning library focused on deep learning.

Another thing to note is how Python manages all these modules through `pip`; Python's default package manager. `Pip` is extremely easy to use and install/uninstall packages with and it is downloaded with Python so no need to smash your head around trying to download Mingw separately for the millionth time. We can simply hit commands like 'pip install pandas' to install a new library.

Dedicated Interactive Environment

Python's Jupyter Notebook is one of the most impactful tools in modern programming, particularly in the fields of data science, machine learning, education, and research. Jupyter has become a standard for interactive development.

- Jupyter allows code to be written, executed, and debugged in blocks called cells.
- You can run individual cells instead of executing the entire codebase, making it easier to test ideas.
- You can combine executable code with formatted text, visuals, and markdown. This enables the creation of detailed, interactive documents that include:
  - Code
  - Explanations
  - Visualizations
  - Formulas
- Jupyter supports over 40 programming languages, not just Python, making it versatile for multi-language projects (making Python more interoperable I suppose).

This entire documentation includes everything from images to code blocks and their outputs to formatted text and hyperlinks, all of it is done in jupyter notebooks.

Applications of Python

Python has established itself as one of the most versatile programming languages, being used across a wide range of industries. This section will highlight some of the major fields where Python is used extensively, along with real-world examples of its applications.

Major Fields Where Python Excels

1. Data Science and Artificial Intelligence (AI)

Python has become the de facto language for data analysis, machine learning, and AI due to its simplicity and the availability of powerful libraries.

- **Libraries Used**: Pandas, NumPy, Scikit-learn, TensorFlow, PyTorch, Matplotlib, Seaborn.
- **Examples**:
  - **Spotify**: Spotify uses Python for its recommendation system, including the famous Wrapped feature that analyzes user data.
  - **Netflix**: Python is a key component of Netflix's machine learning pipeline, helping with content recommendations and predictive analytics.
  - **Uber**: Uber leverages Python for data visualization and geospatial analysis, enabling optimized routing and ride pricing.

2. Web Development

Python's simplicity, along with frameworks like Django and Flask, makes it an excellent choice for developing robust web applications.

- **Frameworks Used**: Django, Flask, FastAPI, Pyramid.
- **Examples**:
  - **Instagram**: Instagram's backend is powered by Python and Django, ensuring scalability and fast feature development.
  - **Reddit**: Reddit is built on Python, allowing the platform to handle massive traffic loads while being easily extensible.
  - **Dropbox**: Dropbox uses Python to manage its server-side functionality and client applications.

3. Automation and Scripting

Python is often the go-to choice for automating repetitive tasks and writing scripts due to its straightforward syntax.

- **Libraries Used**: Selenium, PyAutoGUI, Paramiko, Beautiful Soup.
- **Examples**:
  - **Google**: Python is used at Google extensively. Python scripts automate repetitive tasks, such as data scraping and report generation.
  - **NASA**: NASA employs Python scripts to automate various engineering workflows and simulations.
  - **Excel Automation**: Businesses use Python (via libraries like OpenPyXL) to process and analyze massive Excel files efficiently.

4. Cybersecurity and Ethical Hacking

Python's flexibility and extensive libraries make it ideal for developing penetration testing tools and cybersecurity solutions.

- **Libraries Used**: Scapy, Nmap, Impacket, Requests.

- **Examples**:
  - **Wireshark Extensions**: Wireshark is a free, open-source network analyzer that allows users to capture and analyze network packets in real time. Python is used to write scripts for analyzing packet data at Wireshack.
  - **Automated Vulnerability Scanning**: Security professionals use Python for developing custom scripts for vulnerability testing.

5. Scientific Computing and Research

Researchers widely use Python for simulations, modeling, and analyzing data due to its robust mathematical and statistical libraries.

- **Libraries Used**: SciPy, SymPy, Biopython.
- **Examples**:
  - **Large Hadron Collider (CERN)**: Python is used to analyze particle physics experiments.
  - **SpaceX**: Python supports various simulation tools and real-time decision-making during launches.

Python's Impact in the Real-World

The examples mentioned in the above 4 sections are more then enough to solidify Python as one of the main programming languages shaping the world's technological ecosystem. But just for the sake of completion, lets cover a few more general examples:

- **Google**: Python is a core language at Google, used in applications like YouTube and Google App Engine.
- **Facebook**: Python is utilized for infrastructure management and machine learning pipelines.
- **Instagram**: Leveraging Python's Django framework to maintain high scalability and performance.
- **Amazon**: Python supports Amazon's fulfillment center automation and Alexa's AI engine.
- **Quora**: The platform is built on Python and uses it to scale its user experience.
- **Pinterest**: Pinterest's backend is powered by Python for handling billions of pins.
- **Robinhood**: The financial trading app's backend is built using Python for secure transactions.
- **Quantum Computing**: Libraries like Cirq and Qiskit are enabling Python's entry into quantum programming.
- **Blockchain Development**: Python is used in developing blockchain solutions like smart contracts and cryptocurrency tools.

Python's simplicity, readability, and vast ecosystem of libraries have made it indispensable across numerous domains. Its use by industry leaders like Google, Netflix, and NASA only reinforces its

dominance and applicability in solving modern technological challenges.

Comparison (Python VS C++ VS Java) and Potential Improvements

---

Syntax: Simple and readable. Uses indentation for block structures.

```
def greet():

print("Hello,
World!")

greet()
```

Syntax: Verbose and complex. Uses braces {} for block structures.

```
#include
using namespace
std;
void greet() {
cout << "Hello,
World!" <<
endl;
}


int main() {
greet();
return 0;
}
```

Syntax: Verbose but structured. Also uses braces {} for blocks.

```
public class Main {
public static void
main(String[] args) {
greet();
}
static void greet() {

System.out.println("Hello,
World!");
}

}
```

Performance: Slower execution. Great for prototyping and non-critical tasks.

```
import time
start =
time.time()
total =
sum(range(1,
10000001))
end =
time.time()

print(f"Sum:
{total},
Time: {end -
start}")
```

Performance: Compiled directly to machine code, offering high execution speed.

```
#include
#include
int main() {
auto start =
std::chrono::high_resolution_clock::now();
long total = 0;
for (int i = 1; i <= 10000000; ++i)
total += i;


auto end = std::chrono::high_resolution_clock::now();
std::cout << "Sum: " << total << "\n";
std::cout << "Time: "
          <<
std::chrono::duration_cast<std::chrono::milliseconds>
(end - start).count()
          << " ms\n";
return 0;

}
```

Memory Consumption: Higher due to dynamic typing and abstraction layers.

```
import sys

x = 1000
print(f"Memory
usage of x:
{sys.getsizeof(x)}
bytes")
```

Memory Consumption: Optimized for low-level memory management.

```
#include
int main()
{
int x =
1000;
std::cout
```

Memory Consumption: Garbage collection adds overhead.

```
public class Main {
public static void
main(String[] args) {
int x = 1000;
System.out.println("Memory
usage of x: " +
Integer.BYTES + " bytes");
}
}
```

```
<< "Size of
x: " <<
sizeof(x)
<< "
bytes\n";
return 0;
}
```

Type System: Dynamic typing allows flexibility but increases runtime errors.

```
x = "Hello"
x = 10  #
Allowed,
dynamic
typing
print(x)
```

Type System: Static typing provides stricter error checking.

```
#include

int main() {
int x = 10;
// x =
"Hello";  //
Compile-time
error
std::cout <<
x;
return 0;
}
```

Type System: Static typing with robust error checking.

```
public class Main {
public static void
main(String[] args) {
int x = 10;
// x = "Hello";  //
Compile-time error
System.out.println(x);
}
}
```

Compilation/Execution: Compiled to bytecode and then interpreted by the PVM.

Compilation/Execution: Compiled directly to machine code.

Compilation/Execution: Compiled to bytecode and executed on the JVM.

Compilation/Execution: Compiled to bytecode and then interpreted by the PVM.

Compilation/Execution: Compiled directly to machine code.

Compilation/Execution: Compiled to bytecode and executed on the JVM.

Libraries: Extensive ecosystem with specialized libraries for AI, web, and more.

Libraries: Limited compared to Python. Focuses on system-level programming.

Libraries: Vast libraries for enterprise-grade applications.