Understanding Computer Science

Nithin Vadekkapat

March 8, 2025

What is Computer Science?

- Computer Science is not merely the study of computers.
- Computers are tools that aid in problem-solving.
- The focus is on problems, problem-solving, and algorithmic solutions.

The Role of Algorithms

- An algorithm is a step-by-step set of instructions to solve a problem.
- Some problems may not have a solution.
- Computer Science studies both solvable and unsolvable problems.

Computability in Computer Science

- A problem is **computable** if an algorithm exists to solve it.
- Computer Science studies both computable and non-computable problems.
- Solutions are independent of the machine used.

Abstraction in Computer Science

- Abstraction helps separate the logical and physical perspectives.
- Example: Driving a car.
- Users interact with the interface without needing to understand the mechanics.

Procedural Abstraction

- Users (clients) do not need to know implementation details.
- The interface provides a way to interact with complex implementations.
- Example: Python's `math` module.

```
1 import math
2 math.sqrt(16)
```

- We do not need to know how `sqrt` is implemented.
- We only need to know its name and how to use it.

Programming and Algorithms

- Programming is encoding an algorithm into a notation (programming language) for computer execution
- Without an algorithm, there can be no program
- Computer science is not just programming, but programming is an essential part
- Programming creates representations of our solutions

From Algorithms to Programs

- Algorithms describe solutions in terms of:
  - Data needed to represent the problem instance
  - Steps necessary to produce the intended result
- Programming languages must provide notation for both process and data
- Languages provide control constructs and data types

Control Constructs

- Control constructs allow algorithmic steps to be represented unambiguously
- Minimum requirements for algorithm representation:
  - Sequential processing
  - Selection for decision-making
  - Iteration for repetitive control
- Any language with these basic statements can represent algorithms

Data Types

- All data in computers are strings of binary digits
- Data types provide interpretation for binary data
- Built-in/primitive data types are building blocks for algorithm development
- Example: Integer data type
  - Binary digits interpreted as familiar numbers (23, 654, -19)
  - Supports operations like addition, subtraction, multiplication

Complexity Challenges

- Problems and solutions are often very complex
- Simple language-provided constructs and data types:
  - Are sufficient to represent complex solutions
  - But can be at a disadvantage during problem-solving
- Higher-level abstractions help manage this complexity

Abstraction in Problem-Solving

- Computer scientists use abstractions to focus on the "big picture"
- Creating models of the problem domain enables:
  - More efficient problem-solving process
  - More consistent description of data with respect to the problem
- Abstractions allow us to ignore implementation details temporarily

Data Abstraction

- Abstract Data Type (ADT): logical description of data and operations
- Focuses on *what* the data represents, not *how* it's constructed
- Creates encapsulation around data through information hiding
- Users interact with the interface only, not the implementation

ADT Implementation

- Implementation of an ADT is called a data structure
- Data structure provides the physical view of the data
- Uses programming constructs and primitive data types
- Different implementations can satisfy the same ADT
- Choice of implementation affects efficiency

Objects

- Everything in Python is an object
- Objects have attributes and methods
- Objects can be created from classes
- Classes define the attributes and methods of objects

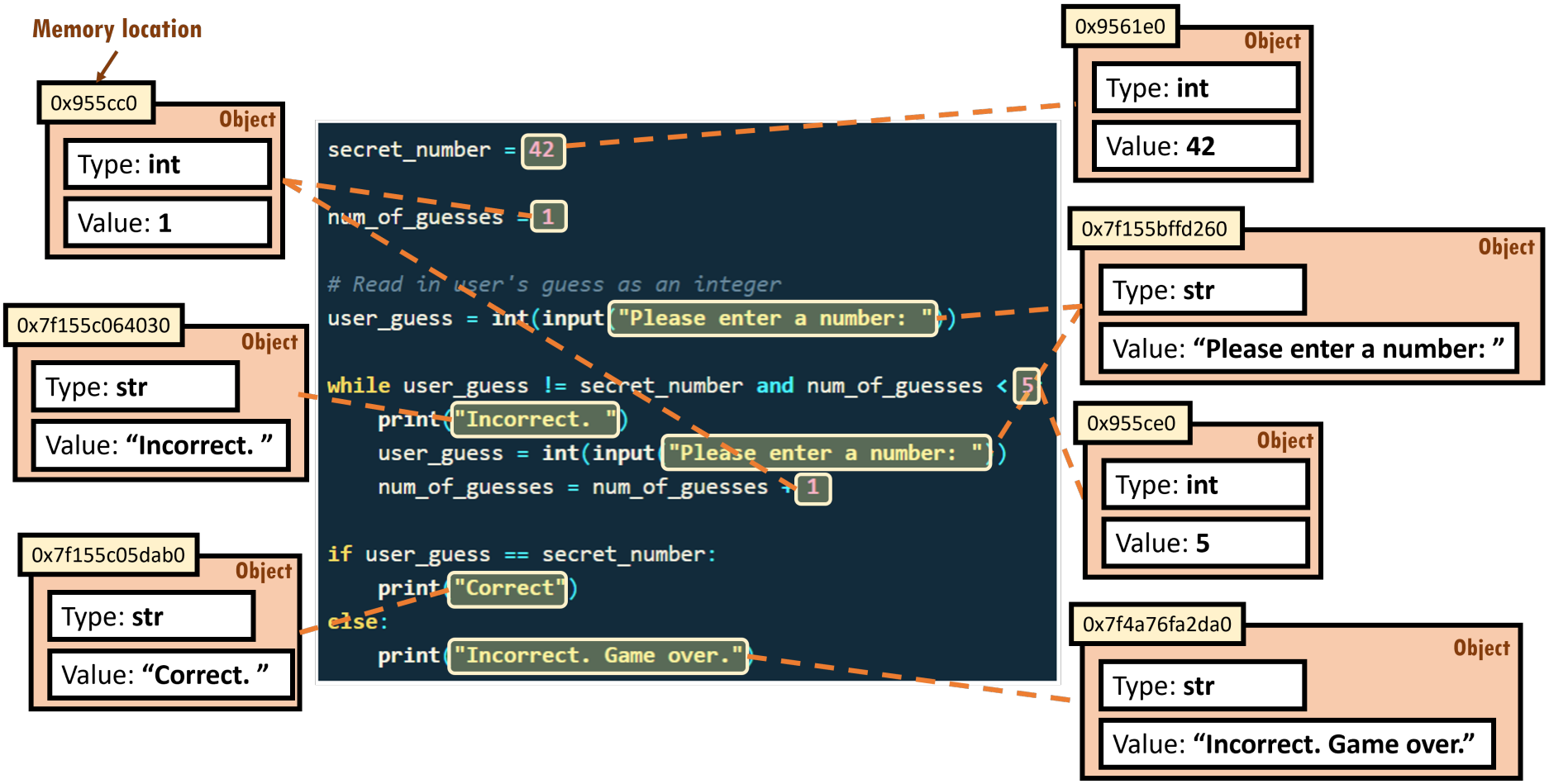

Figure: Objects in Python

Built in atomic datatypes

- Python provides built-in data types
- These types are used to represent data in programs
- Common data types include:
  - Integers
  - Floats
  - Strings
  - Booleans

Operators and Expressions

- Objects can be combined using operators
- Arithmrtic operators in python
  - + (addition)
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - // (floor division, or integer part of division)
  - % (modulo, or remainder after division)
  - ** (exponential - raised to the power)

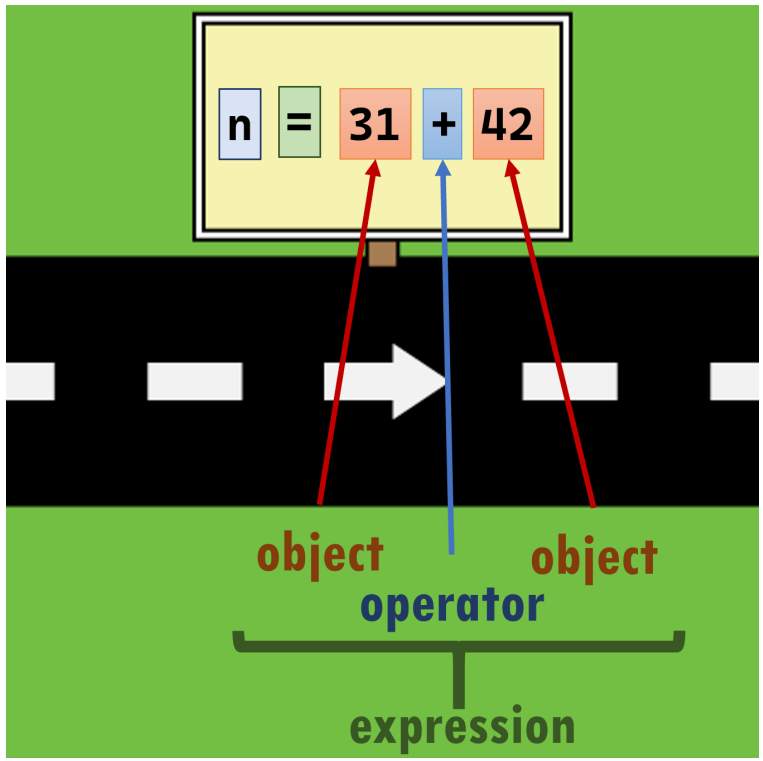

Figure: Operators in Python

Python Operator Precedence

- Python evaluates expressions based on operator precedence.
- Common operators in order of precedence (from highest to lowest):
  - `()` - Parentheses
  - `**` - Exponentiation
  - `+x`, `-x`, `~x` - Unary plus, Unary minus, Bitwise NOT
  - `*`, `/`, `//`, `%` - Multiplication, Division, Floor division, Modulus
  - `+`, `-` - Addition, Subtraction
  - `<<`, `>>` - Bitwise shift operators
  - `&` - Bitwise AND
  - `^` - Bitwise XOR
  - `|` - Bitwise OR
  - `==`, `!=`, `>`, `<`, `>=`, `<=`, `is`, `is not`, `in`, `not in` - Comparisons, Identity, Membership
  - `not` - Logical NOT
  - `and` - Logical AND
  - `or` - Logical OR
- Example: `1 + 2 * 3 == 7` evaluates as `1 + (2 * 3) == 7`, which is `1 + 6 == 7`, resulting in `True`.

Overloaded Operators

- Some operators are not limited to arithmetic operations with numbers.
- For example, + and * can be used for a different kind of operation when used with strings.

```
1 >>> 'hello' + 'world'
2 'helloworld'
3 >>> 'hello' * 3
4 'hellohellohello'
```

Variables

In Python, a variable is simply a name or a label that points to an object instance in memory

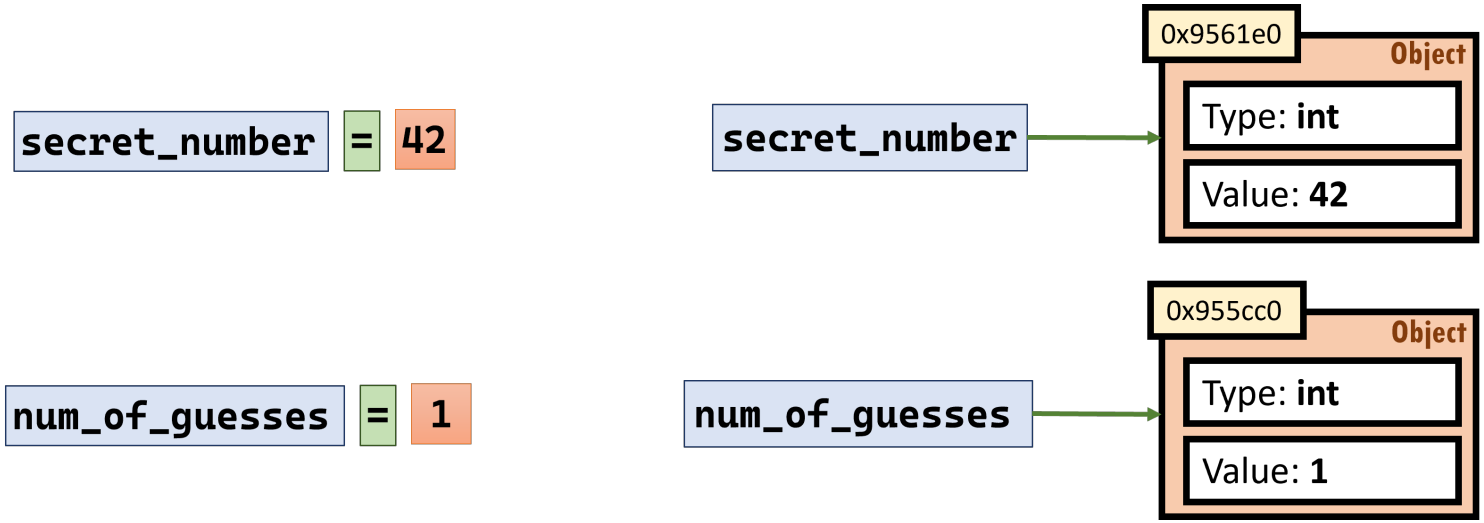

Figure: Variables in Python

Variable Assignments

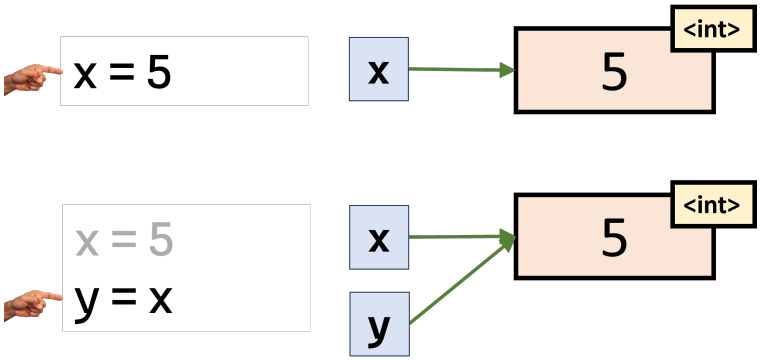

Figure: Variable Assignment

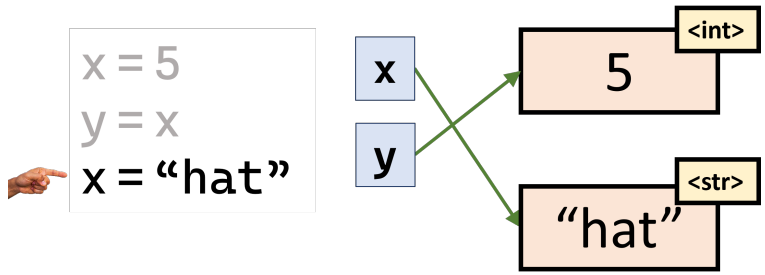

Figure: Variable Assignment

Variable Assignment

```
>>> x = 20
>>> y = x
>>> x = "hello"
>>> print(y)
20
>>> print(x)
hello
```

Naming Variables

- Variable names must start with a letter or an underscore
- The rest of the name can contain letters, numbers, and underscores
- Variable names are case-sensitive
- Variable names should be descriptive

Reserved Keywords

```
>>> import keyword
>>> keyword.kwlist
```

Comparison Operators

- Comparison operators are used to compare two values
- They return a boolean value (True or False)
- Common comparison operators include:
  - == (equal to)
  - != (not equal to)
  - > (greater than)
  - < (less than)
  - >= (greater than or equal to)
  - <= (less than or equal to)

Python Quiz-1

```
>>> 5 == 5.0000
>>> 1+2 == 3
>>> 9 > 8.999999999999999
>>> 6 // 3 == 6.0 / 3.0
```

Floating-Point Comparisons

- Floating-point numbers can have precision issues due to their representation in memory.
- Comparisons involving equality can be affected by these precision issues.
- Example: `1.1 + 2.2 == 3.3` evaluates to `False` due to tiny inaccuracies.
- However, comparisons involving inequality are less likely to be affected.
- Example: `9 > 8.999999999999999` evaluates to `True`.

```
>>> 1.1 + 2.2 == 3.3
False
>>> 9 > 8.999999999999999
True
```

Python Quiz-2

```
>>> 5 != 5.0000
>>> True != False
>>> False < True
>>> "10" > "2"
```

String Comparison

- When comparing two strings, Python compares the "Unicode code point" of the first character of each string.
- If the first characters are the same, it compares the second characters, and so on, until one of the strings ends

String Comparison

- When comparing two strings, Python compares the "Unicode code point" of the first character of each string.
- If the first characters are the same, it compares the second characters, and so on, until one of the strings ends

```
>>> "10" > "2"
False
>>> ord("1")
49
>>> ord("2")
50
```

- This explains why files on your computer might be sorted in this order:
  - `"Picture1.jpg"`
  - `"Picture10.jpg"`
  - `"Picture100.jpg"`
  - `"Picture11.jpg"`
  - `"Picture2.jpg"`

Naming Your Robot

- Use `input()` to read the robot's name from the user.
- Assign the name to a variable called `name`.
- Assign an identifier to the robot, e.g., `identifier = 1000`.
- Print a message from the robot using `print()`.

```
>>> name = input("Enter the robot's name: ")
>>> identifier = 1000
>>> print(f"Hello. My name is {name}. My ID is {
    identifier}.")
```

Converting 2D Index to 1D Index

- User provides the number of rows R, number of columns C, and a 2D index (r, c)
- Convert the 2D index (r, c) to a 1D index s using row-major ordering.

```
1 # Get user input
2 R = int(input("Enter the number of rows: "))
3 C = int(input("Enter the number of columns: "))
4 r = int(input("Enter the row index: "))
5 c = int(input("Enter the column index: "))
6 s = r*R + c
7 # Print the result
8 print(f"The 1D index is: {s}")
```

What are Chained Comparisons?

- Python allows multiple comparison operators to be chained together in a single expression.
- This creates a more readable and concise way to express complex comparisons.
- The comparisons are evaluated from left to right, and they are implicitly combined using the logical `and` operator.

Example 1: Simple Range Check

- Consider checking if a variable `x` is within a specific range.
- Without chained comparisons:

```
1  if 10 < x and x < 20:
2      # ...
```

- With chained comparisons:

```
1  if 10 < x < 20:
2      # ...
```

- Both are equivalent, but the chained version is more readable.

Example 2: Multiple Comparisons

```
1  x = 5
2  y = 10
3  z = 15
4  if x < y < z:
5      print("x < y < z is True")
```

- This checks if `x` is less than `y` AND `y` is less than `z`.

Example 3: Equality and Inequality

```
1  a = 10
2  b = 10
3  c = 15
4  if a == b < c:
5      print("a==b<c is True")
```

- This checks if `a` is equal to `b` AND `b` is less than `c`.

Control Structures

- Control structures allow you to control the flow of your program.
- They include:
  - Conditional statements
  - Loops
  - Functions

Conditional Statements

- Conditional statements allow you to execute different blocks of code based on certain conditions.
- The most common conditional statement is the `if` statement.
- The `if` statement evaluates a condition and executes a block of code if the condition is `True`.

If Statement Syntax

```
1 if condition:
2     # code block
```

- The `condition` is an expression that evaluates to `True` or `False`.
- If the `condition` is `True`, the code block is executed.
- If the `condition` is `False`, the code block is skipped.

If Statement Example

```
1 x = 10   # Assign a value to x
2 if x > 5:   # Check if x is greater than 5
3     print("x is greater than 5")   # Print a message
```

- In this example, the condition `x > 5` is `True` because `x` is 10.
- The code block is executed, and the message `"x is greater than 5"` is printed.

If-Else Statement Syntax

```
1 if condition:
2     # code block 1
3 else:
4     # code block 2
```

- If the `condition` is `True`, `code block 1` is executed.
- If the `condition` is `False`, `code block 2` is executed.

If-Else Statement Example

```
1 x = 3  # Assign a value to x
2 if x > 5:  # Check if x is greater than 5
3     print("x is greater than 5")  # Print a message
4 else:  # If x is not greater than 5
5     print("x is not greater than 5")  # Print a
          message
```

- In this example, the condition `x > 5` is `False` because `x` is 3.
- The code block after the `else` statement is executed, and the message `"x is not greater than 5"` is printed.

Nested If Statements

- You can nest `if` statements inside other `if` statements.
- This allows you to check multiple conditions in sequence.
- The inner `if` statement is only executed if the outer `if` statement's condition is `True`.

Nested If Statements Example

```
1 x = 10  # Assign a value to x
2 if x > 5:  # Check if x is greater than 5
3     if x < 15:  # Check if x is less than 15
4         print("x is between 5 and 15")  # Print a
              message
```

- In this example, the condition `x > 5` is `True` because `x` is 10.
- The inner `if` statement checks if `x` is less than 15, which is `True`.
- The message `"x is between 5 and 15"` is printed.

If-Elif-Else Statement Syntax

```
1 if condition1:
2     # code block 1
3 elif condition2:
4     # code block 2
5 else:
6     # code block 3
```

- If `condition1` is `True`, `code block 1` is executed.
- If `condition1` is `False` and `condition2` is `True`, `code block 2` is executed.
- If both `condition1` and `condition2` are `False`, `code block 3` is executed.

Loops

- Loops allow you to repeat a block of code multiple times.
- The two most common types of loops are:
  - `for` loops
  - `while` loops

For Loop Syntax

```
1 for item in iterable:
2     # code block
```

- The `for` loop iterates over each `item` in the `iterable`.
- The `code block` is executed once for each `item` in the `iterable`.

For Loop Example

```
1 for i in range(5):
2     print(i)
```

- In this example, the `range(5)` function generates a sequence of numbers from 0 to 4.
- The `for` loop iterates over each number in the sequence and prints it.

While Loop Syntax

```
1 while condition:
2     # code block
```

- The `while` loop executes the `code block` as long as the `condition` is `True`.
- The `condition` is evaluated before each iteration of the loop.

While Loop Example

```
1 i = 0
2 while i < 5:
3     print(i)
4     i += 1
```

- In this example, the `while` loop prints the value of `i` as long as `i` is less than 5.
- The value of `i` is incremented by 1 in each iteration.