Lecture 1: Introduction to Programming with Python

Gereltsetseg Altangerel

Advisor: Máté Tejfel

Programming Languages and Compilers,
Eötvös Loránd University(ELTE), Budapest, Hungary

gereltsetseg@inf.elte.hu

Course Structure

- **Weeks 1 - 6: Foundations of Python**
  - Python Basics
  - Object-Oriented Programming (OOP)
  - Testing and Debugging
  - Miscellaneous Topics

- **Weeks 8 - 13: Applying Python in Specific Domains**
  - Python in Data Science
  - Python in Web Development: Creating a Simple API
  - Python in Automation

Lecture 1: Overview

1. Programming Fundamentals

2. Programming with Python
   - Python Programs(Scripts) and execution
   - Some Objects, Variables, Operations

1.Fundamentals of Programming

Fundamentals of Programming

- **What Does a Computer Do?**

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions

Fundamentals of Programming

- **What Does a Computer Do?**
    - Performs instructions
        - A billion instructions per second!

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results
    - Hundreds of gigabytes of storage!

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results
    - Hundreds of gigabytes of storage!
- **What kinds of instructions?**

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results
    - Hundreds of gigabytes of storage!
- **What kinds of instructions?**
  - **Built-in instructions:** basic operations like storing information, manipulating data, or performing arithmetic operations.

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results
    - Hundreds of gigabytes of storage!
- **What kinds of instructions?**
  - **Built-in instructions:** basic operations like storing information, manipulating data, or performing arithmetic operations.
  - **Custom instructions:** Ones that you define as the programmer

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results
    - Hundreds of gigabytes of storage!
- **What kinds of instructions?**
  - **Built-in instructions:** basic operations like storing information, manipulating data, or performing arithmetic operations.
  - **Custom instructions:** Ones that you define as the programmer

Computers only know what you tell them.

Fundamentals of Programming

Types of Knowledge

- **Declarative Knowledge (Goal)**: *WHAT WE WANT TO GET FROM COMPUTER.*
  - Statements of fact
  - Example: Get to know each other more.
- **Imperative Knowledge (Steps to reach goal)**: TELLS THE COMPUTER HOW TO DO THINGS.
  - *A recipe or ”how-to”*
  - Example:
    1. Pair up with someone.
    2. Once you find a partner, introduce yourselves to each other within 10 minutes.
    3. Each person has 5 minutes to gather as much information about their partner as possible.
    4. Introduce your partner to the class in 3 minutes.

A Numerical Example

Statement of fact

- The square root of a number x (e.g., 16) is y such that $y \times y = x$, or mathematically, $\sqrt{x} = y$.

Recipe for deducing the square root of a number x :

1. Start with a guess, g.
2. If $g \times g$ is close enough to x, stop and say g is the answer.
3. Otherwise, make a new guess by averaging g and $\frac{x}{g}$.
4. Using the new guess, repeat the process until close enough.

g	$g \times g$	$\frac{x}{g}$	$\frac{g+\frac{x}{g}}{2}$
3	9	$\frac{16}{3} \approx 5.33$	$\frac{3+5.33}{2} \approx 4.17$
4.17	17.36	$\frac{16}{4.17} \approx 3.837$	$\frac{4.17+3.837}{2} \approx 4.0035$
4.0035	16.0277	$\frac{16}{4.0035} \approx 3.997$	$\frac{4.0035+3.997}{2} \approx 4.000002$

Table 1: Iterations for Finding Square Root of 16

What a Recipe Can Consist Of

1. sequence of Simple Steps: A clear and ordered list of instructions.
2. flow of control process that specifies when each step is executed
3. a means of determining when to stop

Algorithm = Step 1 + Step 2 + Step 3

What a Recipe Can Consist Of

1. sequence of Simple Steps: A clear and ordered list of instructions.
2. flow of control process that specifies when each step is executed
3. a means of determining when to stop

Algorithm = Step 1 + Step 2 + Step 3

Does the computer understand this algorithm?

What a Recipe Can Consist Of

1. sequence of Simple Steps: A clear and ordered list of instructions.
2. flow of control process that specifies when each step is executed
3. a means of determining when to stop

Algorithm = Step 1 + Step 2 + Step 3

Does the computer understand this algorithm?

- Algorithms must be written in a high-level programming language like Python.

What a Recipe Can Consist Of

1. sequence of Simple Steps: A clear and ordered list of instructions.
2. flow of control process that specifies when each step is executed
3. a means of determining when to stop

Algorithm = Step 1 + Step 2 + Step 3

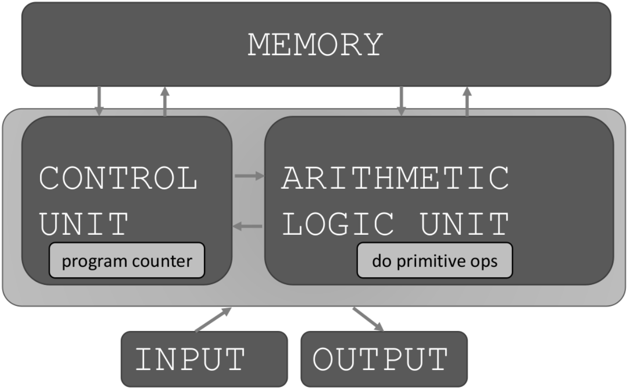
Does the computer understand this algorithm?

- Algorithms must be written in a high-level programming language like Python.
- The computer understands and executes the code with the help of a special program called an interpreter or a compiler: These programs convert high-level code into machine code.

Basic Computer Architecture

The machine stores and executes instructions from a program.

- Programs are stored in the computer's memory.
- Instructions are fetched from memory and executed sequentially by the CPU.



Key Points

- **Algorithm**: A step-by-step procedure or set of rules designed to solve a specific problem or perform a particular task.
- To implement an algorithm, we use a high-level programming language that converts human-readable instructions into something a computer can execute.
  - A programming language allows us to write clear, structured, and precise instructions for the computer.
  - The code that implements an algorithm is called the **source code**.
- The source code is executed on a computer using a special program called a **compiler** or an **interpreter**.
  - A **compiler** translates the entire source code into machine code before execution.
  - An **interpreter** translates and executes the source code line by line.

2. Programming with Python

Programming with Python

- How to write instructions in Python.
- The Python instructions can be categorized into:
  - **Definitions**: Define structures like variables, functions, or classes. These are evaluated when the program runs.
    - Example: `x = 10`, `def my_function():`
  - **Expressions**: Produce values when evaluated.
    - Example: `x + y`, `len("Python")`
  - **Commands**: Execute actions such as controlling the flow of the program or interacting with the user.
    - Example: `print("Hello, World!")`, `if x > 10:`
- These elements instruct the Python interpreter to perform specific tasks, making up the logic and behavior of the program.

What Makes a Language?

Natural Language

- **Alphabet:**
  - Letters (A-Z), numerals, punctuation
- **Lexis:**
  - Words, idioms, expressions
- **Syntax:**
  - Grammar rules
- **Semantics:**
  - Meaning of sentences, context

Programming Language

- **Alphabet:**
  - Characters (A-Z, 0-9, special symbols like `#` `$` `_` `{` `}` `;` `:` etc.)
- **Lexis:**
  - Keywords, identifiers, operators
- **Syntax:**
  - Rules for writing valid code (e.g., loops, conditionals)
- **Semantics:**
  - What the code does when executed

Compiler vs Interpreter

Compiler

- Translates entire source code into machine code at once.
- Produces an independent executable.
- Faster execution (pre-translated).
- Examples: C, C++, Java, Rust.
- Errors detected before execution.

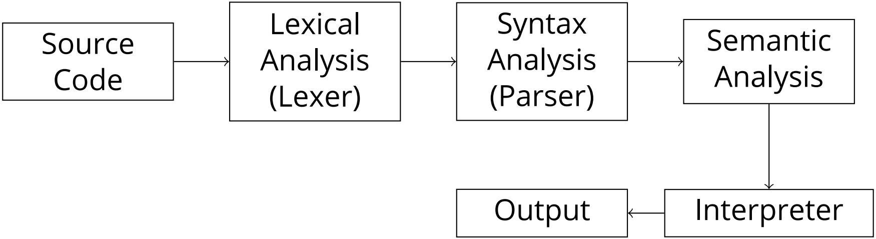
Interpreter

- Translates and runs code line by line.
- Requires source code for execution.
- Slower execution (translates at runtime).
- Examples: Python, Ruby, JavaScript, PHP.
- Errors detected during execution.

Summary

- Source Code -> Compiler ->Executable File -> Execution
- Source Code -> Interpreter -> Line-by-line execution

Python Interpreter



Source Code

Lexical Analysis (Lexer)

Syntax Analysis (Parser)

Semantic Analysis

Interpreter

Output

Data Objects in Python

- **Data objects** are the fundamental units that store and manipulate data in a program.
- Every object has a **type**, which defines the operations and behaviors it supports.
    - **Example:**
        - *Anna* is a human, so she can walk and speak English.
        - *A cat* is an animal, so it can walk and meow.
- **Objects are classified into two main types:**
    - **Scalar objects**:
        - Represent a single, indivisible value (atomic).
        - Cannot be broken down into smaller components.
        - *Example*: integers, floats, booleans.
    - **Non-scalar objects**:
        - Represent collections or sequences of values.
        - Can be broken down into individual elements with accessible internal structures.
        - *Example*: lists, strings, dictionaries.

Scalar Objects

- **int** – represent integers, e.g., 5
- **float** – represent real numbers, e.g., 3.27
- **bool** – represent Boolean values `True` and `False`
- **NoneType** – special type with a single value, `None`
- You can use `type()` to see the type of an object
- These basic scalar types form the **building blocks** for more complex data structures and operations in Python.

Type Conversions (Casting)

- Changing an object from one type to another.
- **Explicit Type Conversion**:
  - **Definition**: Manually converting an object from one type to another using built-in functions.
  - **Examples:**

    `float(3)` converts the integer `3` to `3.0`.

    `int(3.9)` converts the float `3.9` to the integer `3`, truncating the decimal part.
- **Implicit Type Conversion**:
  - **Definition**: Automatic conversion of types by Python during operations to maintain consistency.
  - **Example: Adding an integer to a float**

    `5 + 3.2` results in `8.2`, with the integer `5` being implicitly convert to a float.

Operators on `int` and `float`

- **Addition and Subtraction:**
  - `i + j` → Sum of `i` and `j`
  - `i - j` → Difference between `i` and `j`
- **Multiplication:**
  - `i * j` → Product of `i` and `j`
- **Division and Modulus:**
  - `i / j` → Division result (always `float`)
  - `i % j` → Remainder when `i` is divided by `j`
- **Exponentiation:**
  - `i ** j` → `i` raised to the power of `j`

Expressions

- Expressions: Combinations of objects (values) and operators that produce a result.
- Each expression has a **value** which is the result of the expression, and this value has a specific **type**.
- Syntax of a Simple Expression:
  - `<object> <operator> <object>`
- Examples:
  - `3 + 5`
  - `7.5 * 2`
  - `10 / 4`

Type of Result:

- If both objects (operands) are `int`, the result is `int`.
- If either or both objects (operands) are `float`, the result is `float`.

Question: What is the result of `5 % 3`?

- **Compute:** `5 % 3`
- What do you think the result is?

Question: What is the result type of the following expressions?

- **Expressions:**
  - `type(3 + 5)`
  - `type(7.5 - 5)`
- **What are the result types of these expressions?**

Operator Precedence and Execution Order

- Parentheses are used to tell Python to do these operations first.
- Operator precedence without parentheses:
    - $**$
    - $*$
    - $/$
    - $+$ and $-$ (executed left to right, as they appear in the expression)

Operator Precedence Examples

- Expression 1:

```
3 * 2 ** 2
```

Operator Precedence Examples

- Expression 1:

  ```
  3 * 2 ** 2
  ```

  Result: `12`

- Expression 2:

  ```
  2 + 3 * 4
  ```

Operator Precedence Examples

- Expression 1:

  ```
  3 * 2 ** 2
  ```

  Result: `12`
- Expression 2:

  ```
  2 + 3 * 4
  ```

  Result: `14`
- Expression 3:

  ```
  (2 + 3) * 4
  ```

Operator Precedence Examples

- Expression 1:

  ```
  3 * 2 ** 2
  ```

  Result: `12`

- Expression 2:

  ```
  2 + 3 * 4
  ```

  Result: `14`

- Expression 3:

  ```
  (2 + 3) * 4
  ```

  Result: `20`

Variable

- A variable is a name that refers to a value stored in memory.
- Creating variable - Binding variable name and value - The equal sign ('=') is used to assign a value to a variable name.
  - `pi = 3.14159`
  - `pi_approx = 22/7`
- The value is stored in computer memory.
- The variable name acts as a reference to this value.
- You can retrieve the value by simply typing the variable name, like `pi`.
- Good variable names improve both readability and maintainability.

Why give names to values?

```
pi = 3.14159
radius = 2.2
# Calculate the area of a circle
area = pi * (radius**2)
```

Why give names to values?

- To reuse names instead of values.

```
pi = 3.14159
radius = 2.2
# Calculate the area of a circle
area = pi * (radius**2)
```

Why give names to values?

- To reuse names instead of values.
- Makes it easier to change code later.

```
pi = 3.14159
radius = 2.2
# Calculate the area of a circle
area = pi * (radius**2)
```

The Importance of Meaningful Variable Names

Code Readability:

- Code is not only read by the computer but also by people.
- Choose variable names that clearly describe their purpose.
- Meaningful names improve code maintainability and collaboration.

```
a = 3.14159
b = 2.2
c = a * (b**2)
```

```
pi = 3.14159
radius = 2.2
area = pi * (radius**2)
```

Variable naming rules

- Can include uppercase and lowercase letters, digits, and the underscore _.
- Cannot start with a digit.
- Variable names are case-sensitive.
  - Example: `Romeo` and `romeo` are different variables.
- Cannot use Python keywords or reserved words.
  - Example: `if`, `while`, `return` are not allowed as variable names.

Which one is valid variable names?

1. `my_variable`, `_count1`, `TotalSum`
2. `1stPlace`, `for`

Multiple binding (assignment)

- Python allows multiple assignments. The statement

```
x, y = 2, 3
```

What will it print?

```
x, y = 2, 3
x, y = y, x
print('x =', x)
print('y =', y)
```

Multiple binding (assignment)

- Python allows multiple assignments. The statement

```
x, y = 2, 3
```

What will it print?

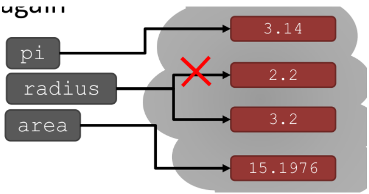
```
x, y = 2, 3
x, y = y, x
print('x =', x)
print('y =', y)
```

It prints
x = 3
y = 2

Changing Bindings

- Variables can be re-bound using new assignment statements.
- The previous value may still be stored in memory but is no longer accessible through the variable name.
- The value for `area` does not change until you tell the computer to perform the calculation again.

```
pi = 3.14159
radius = 2.2
area = pi * (radius**2)
radius = radius + 1
```



Literal, Object, Variable

```
                            +------------------+
                            |     Source Code  |
                            +------------------+
                            |   (literal: 2.2)  |
                            |    radius = 2.2    |
                            +------------------+
                                       |
                                       | creates
                                       v
+----------------+          +--------------------+
|  Variable      |  ------> |  Float Object      |
|  (radius)      |  points  |  (Value: 2.2)      |
+----------------+          +--------------------+
```

Non-Scalar Object: String in Python

- Letters, special characters, spaces, digits
- Enclose in double or single quotes:

```
hi = "hello there"
```

- Concatenate strings:

```
name = "Anna"
greet = hi + name
greeting = hi + " " + name
```

- Perform operations on strings as defined in Python docs:

```
silly = hi + " " + name * 3
```

Comparison Operators

- Comparison operators are used to evaluate expressions and return a Boolean result (True or False).
- Variables *i* and *j* can be of type `int`, `float`, or `string`.
- $i > j$
- $i \geq j$
- $i < j$
- $i \leq j$
- $i == j$ *equality test: True if i is the same as j.*
- $i \neq j$ *inequality test: True if i is not the same as j.*

Logic Operators on Booleans

a and *b* are variables with Boolean values.

- `not a`
  True if a is False, False if a is True.
- `a and b`
  True if both a and b are True.
- `a or b`
  True if either or both a and b are True.

a	*b*	*a* and *b*	*a* or *b*
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Comparison Example

```
score = 85
passing_score = 70
print(score >= passing_score)

is_student = True
has_permission = False
can_enter = is_student or has_permission
print(can_enter)
```

Statically Typed vs. Dynamically Typed Languages

- **Statically Typed Languages:**
  - Type Declaration: Must declare the type of each variable.
  - Example (Java):

```
int number = 5;
number = "Hello";
```

This will cause a compile-time error because "Hello" is not an integer.

- **Dynamically Typed Languages:**
  - Type Inference: Types are inferred at runtime; no need for explicit declaration.
  - Example (Python):

```
number = 5
number = "Hello"
```

Here, 'number' starts as an integer and later changes to a string without error.

- **Summary:**
  - **Statically Typed**: Types are fixed and checked at compile time.
  - **Dynamically Typed**: Types are flexible and checked at runtime.

Key Points in Programming with Python

- We will learn how to write instructions in Python to solve specific problems.
- To create meaningful instructions, we need to understand Python's rules and language elements.
- We explored data objects in Python, including:
    - **Scalar Objects**: Indivisible values such as `int`, `float`, `bool`.
    - **Non-scalar Objects**: Collections or sequences of values such as `list`, `string`, `dict`.
- We discussed typecasting and Python's dynamic typing, which allows variables to change types based on the values assigned to them.

Thank You for Your Attention!