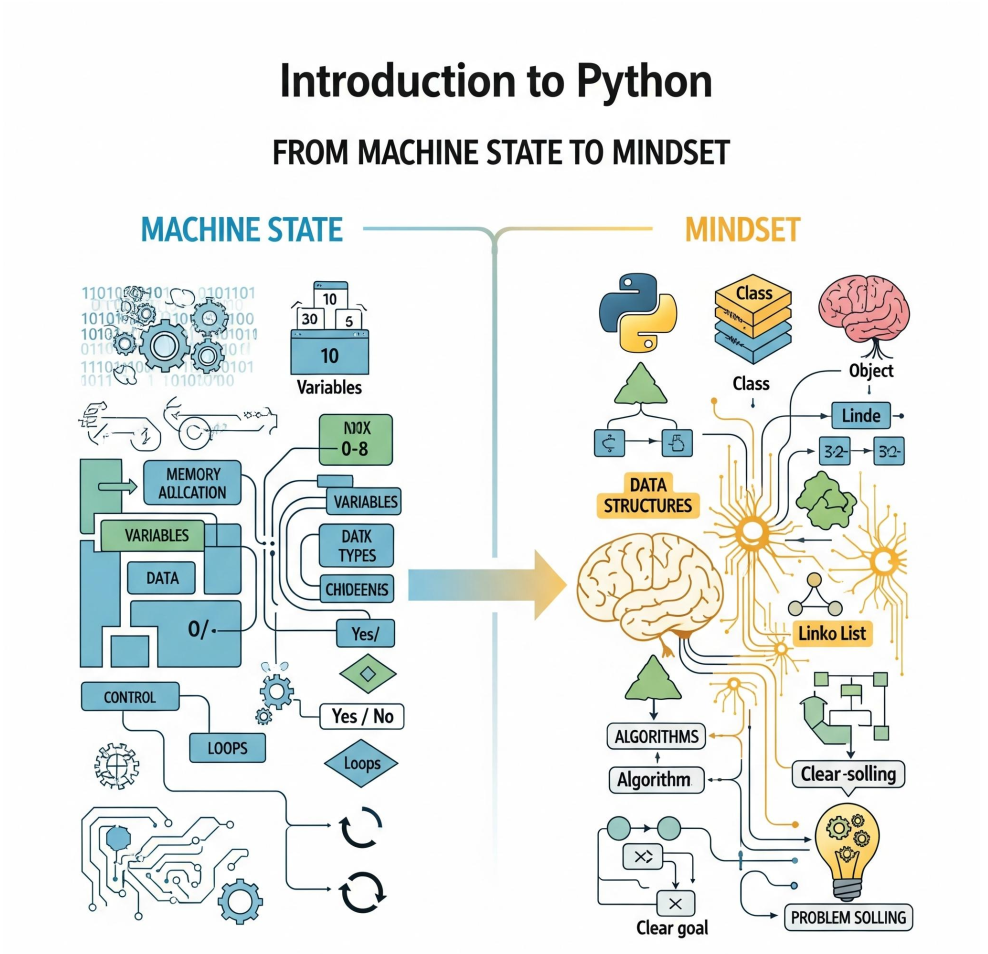
Lecture - 2

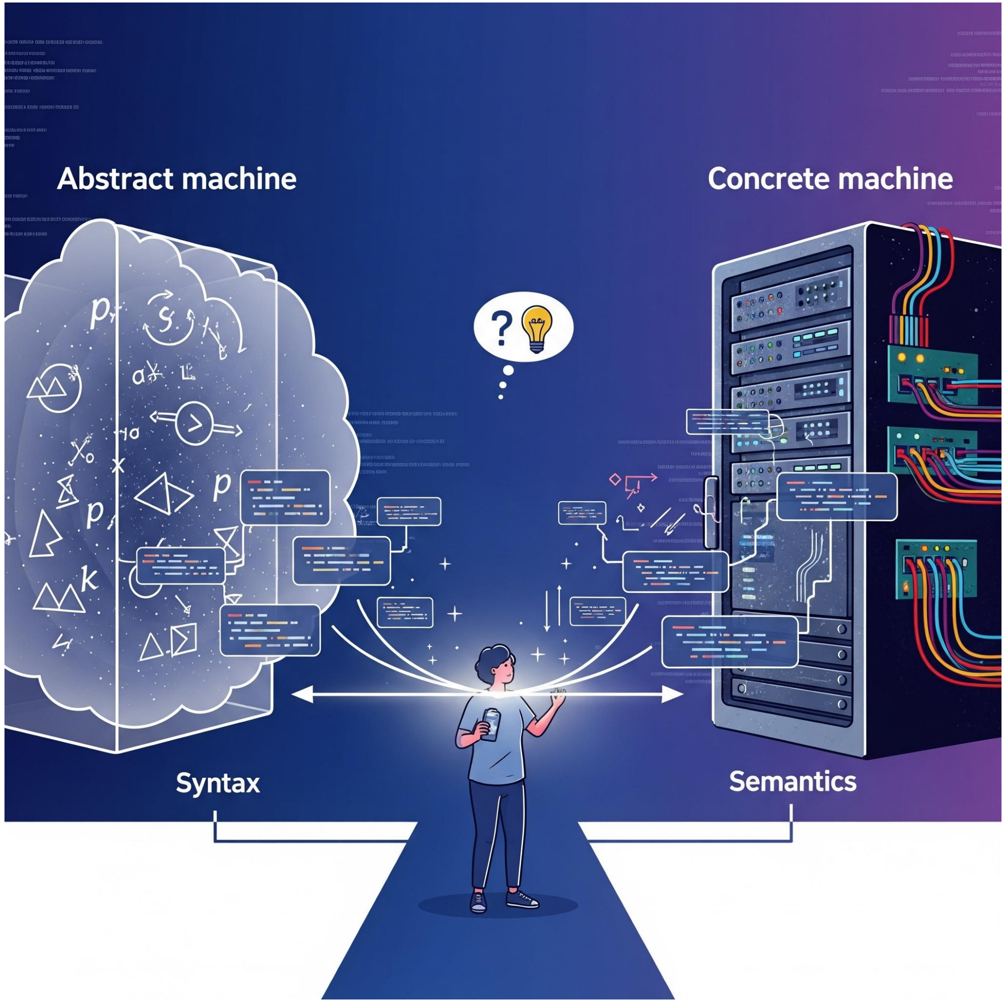
The Building Blocks - Syntax, Semantics, and Control Flow

"How a Programming Language is Formally Defined ? "



Lecture Recap

- Differentiate between **syntax** and **semantics**
- Understand **concrete** versus **abstract** machines
- Explore roles of code **creator** and **validator**



Defining Python Programming Language

01 **Syntax** defines valid Python program structure

02 **Core semantics** describes program execution effects

03 **Python Language Reference** is the *definitive source*

https://docs.python.org/3/reference/index.html

04 **CPython** is the original and most maintained implementation

Python Language Reference

01 Python Language Reference defines syntax and core semantics

02 **Ambiguities** may exist due to informal descriptions

03 Written in English for better understanding by readers

04 **Formal specifications** are primarily for syntax

Python Language Reference
(Definitive Source for Syntax and Semantics)

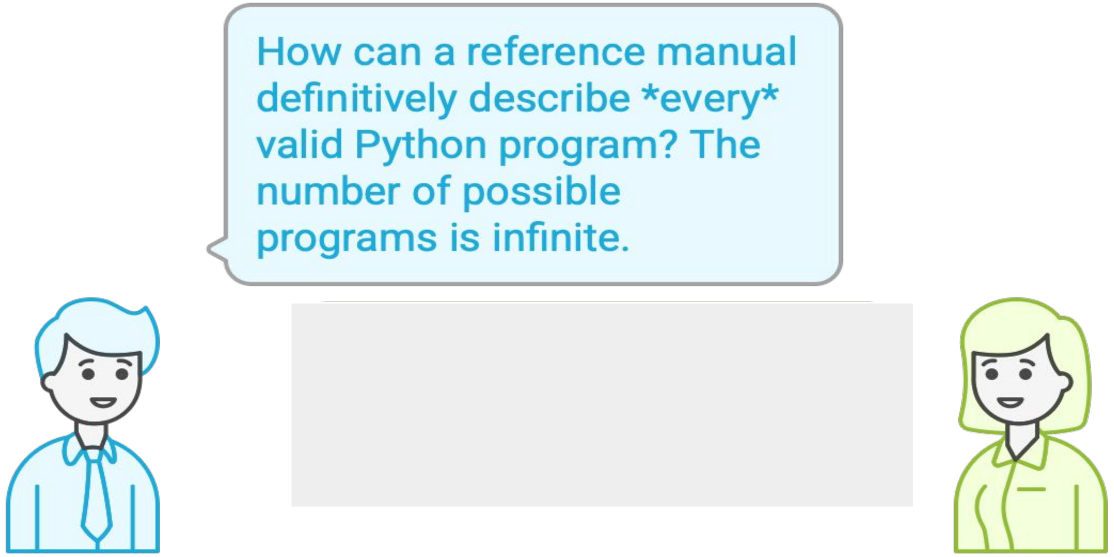
“While I am trying to be as precise as possible, I chose to use **English rather than formal specifications** for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will l**eave room for ambiguities**.”

“On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here.” — COMPLETENESS Assurance.

“If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).” — Trend in PL Research (Use of Theorem Provers)

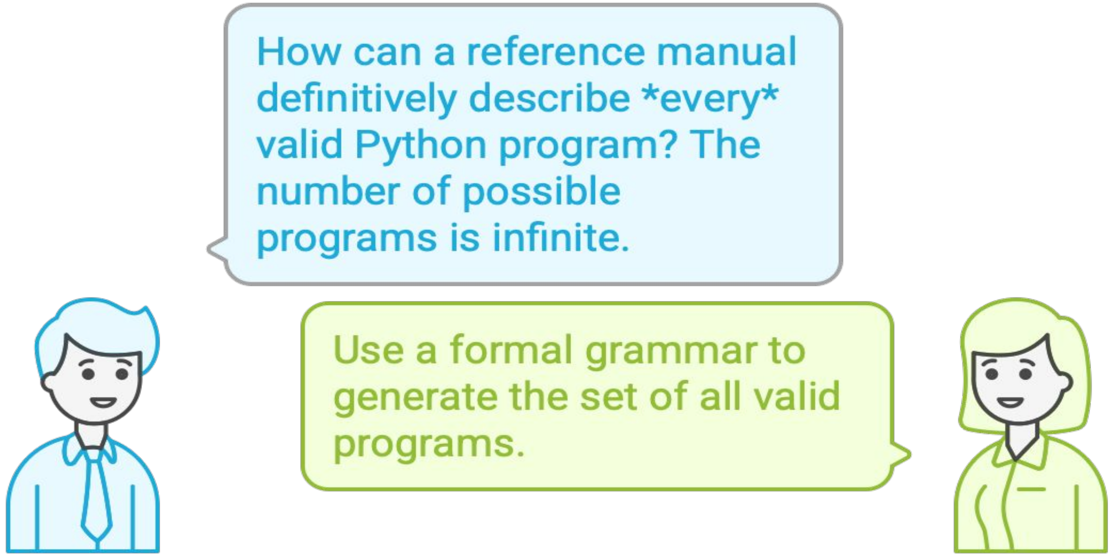
Python Language Reference

Describing Infinite Programs with Finite Rules


How can a reference manual definitively describe *every* valid Python program? The number of possible programs is infinite.

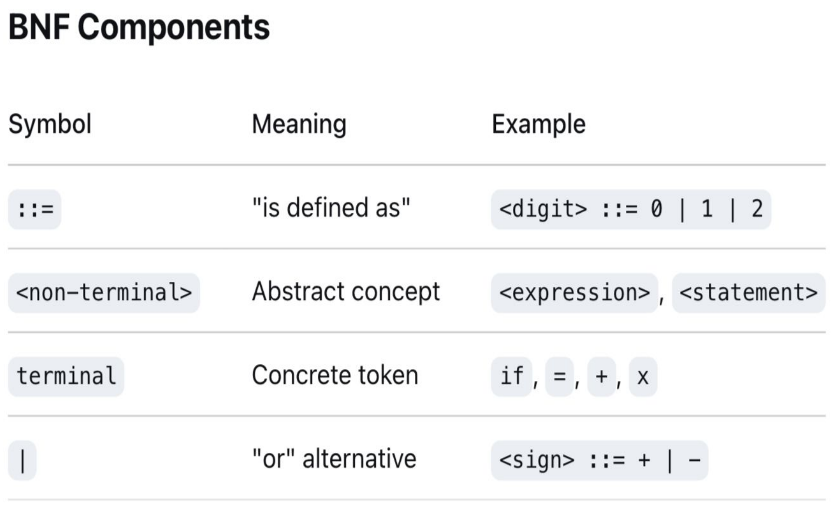
Python Language Reference

Describing Infinite Programs with Finite Rules


How can a reference manual definitively describe *every* valid Python program? The number of possible programs is infinite.
Use a formal grammar to generate the set of all valid programs.

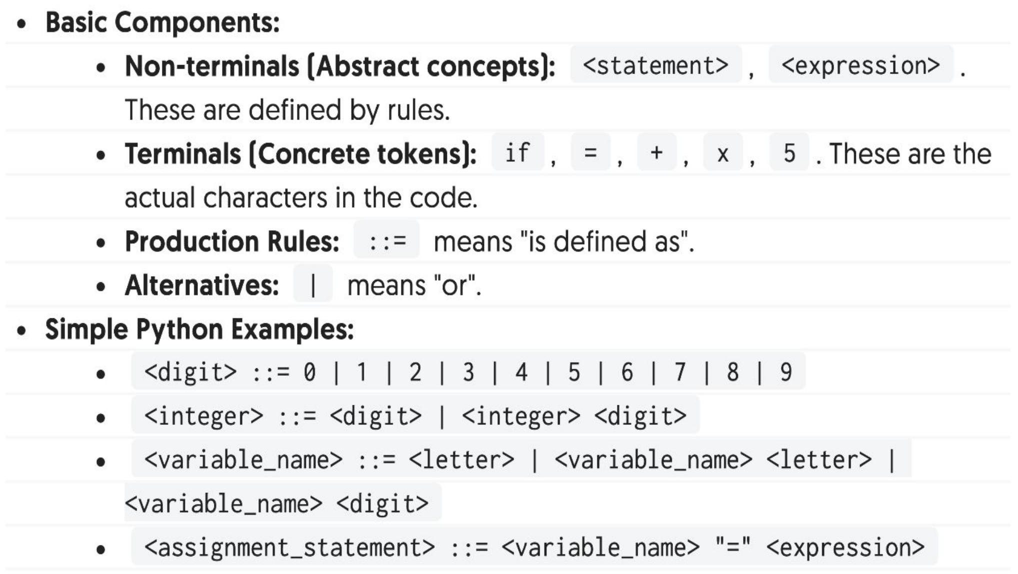
Formal Syntax - Backus-Naur Form (BNF)

Characters → Tokens → Expressions → Statements → Programs



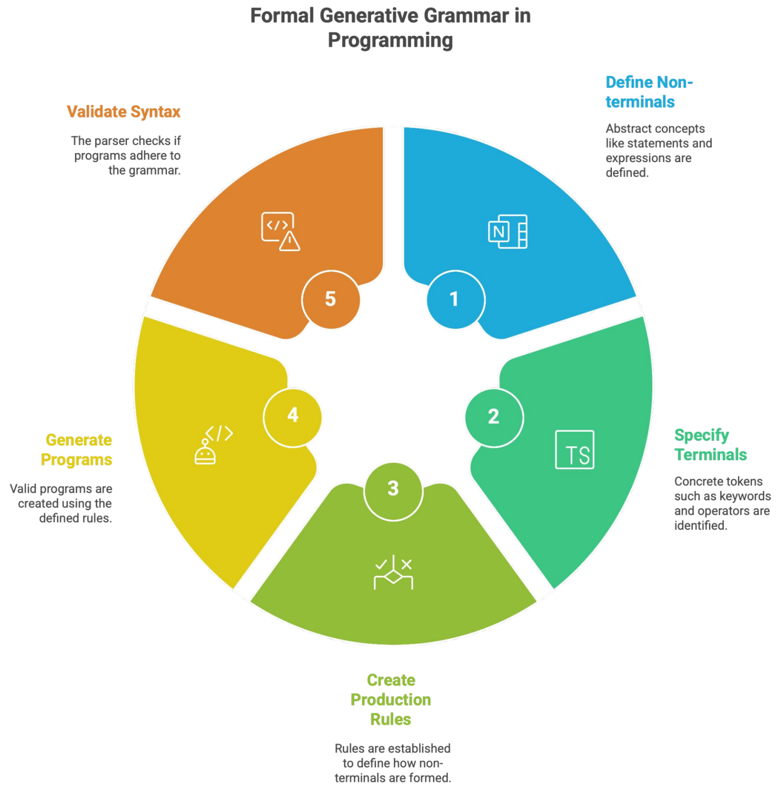
BNF Components

Symbol	Meaning	Example
`::=`	"is defined as"	`<digit> ::= 0 | 1 | 2`
`<non-terminal>`	Abstract concept	`<expression>`, `<statement>`
`terminal`	Concrete token	`if`, `=`, `+`, `x`
`|`	"or" alternative	`<sign> ::= + | -`



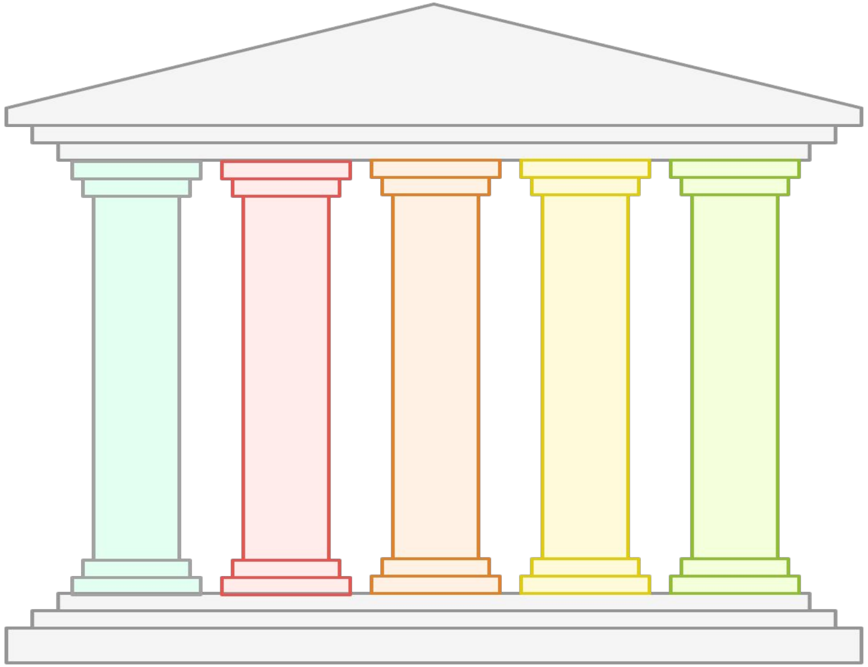
- **Basic Components:**
  - **Non-terminals (Abstract concepts):** `<statement>` , `<expression>` . These are defined by rules.
  - **Terminals (Concrete tokens):** `if` , `=` , `+` , `x` , `5` . These are the actual characters in the code.
  - **Production Rules:** `::=` means "is defined as".
  - **Alternatives:** `|` means "or".
- **Simple Python Examples:**
  - `<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
  - `<integer> ::= <digit> | <integer> <digit>`
  - `<variable_name> ::= <letter> | <variable_name> <letter> | <variable_name> <digit>`
  - `<assignment_statement> ::= <variable_name> "=" <expression>`

Formal Syntax - Backus-Naur Form (BNF)


Formal Generative Grammar in Programming
Define Non-terminals
Abstract concepts like statements and expressions are defined.
1
Specify Terminals
Concrete tokens such as keywords and operators are identified.
2
Create Production Rules
Rules are established to define how non-terminals are formed.
3
Generate Programs
Valid programs are created using the defined rules.
4
Validate Syntax
The parser checks if programs adhere to the grammar.
5

- **Basic Components:**
  - **Non-terminals (Abstract concepts):** `<statement>`, `<expression>`. These are defined by rules.
  - **Terminals (Concrete tokens):** `if`, `=`, `+`, `x`, `5`. These are the actual characters in the code.
  - **Production Rules:** `::=` means "is defined as".
  - **Alternatives:** `|` means "or".
- **Simple Python Examples:**
  - `<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
  - `<integer> ::= <digit> | <integer> <digit>`
  - `<variable_name> ::= <letter> | <variable_name> <letter> | <variable_name> <digit>`
  - `<assignment_statement> ::= <variable_name> "=" <expression>`

Variables, Keywords, and Types



01 **Variables and Identifiers**

Rules and conventions for naming variables and constants in Python.

Keywords

Reserved words with special meanings that cannot be used as variable names.

Dynamic Typing

Variables can be assigned to any type of object at runtime.

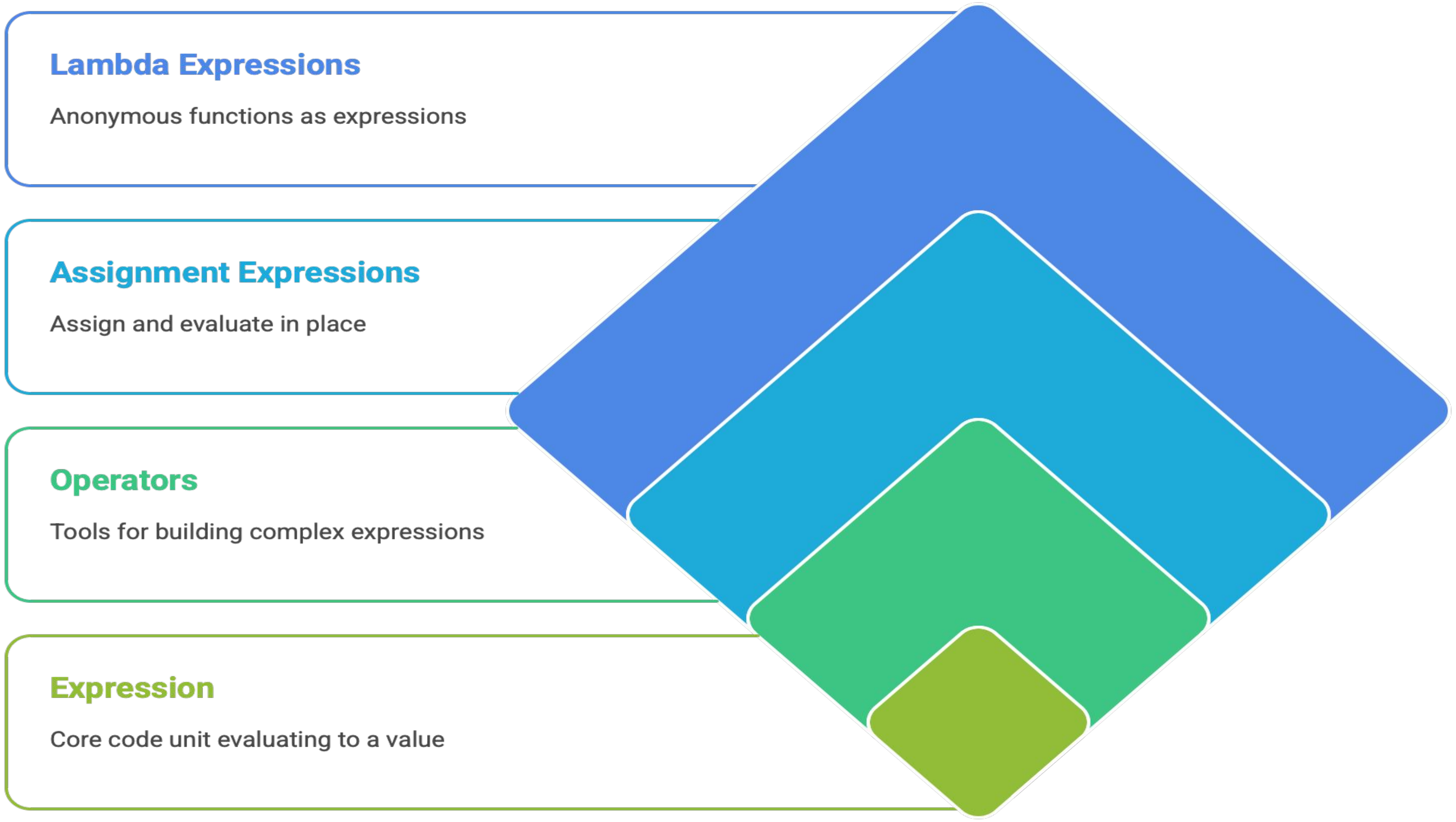
Strong Typing

Strict type-checking ensures operations are valid for the object's type.

Optional Static Typing

Type hints for clarity and tooling, ignored by the interpreter at runtime.

Python Expression Hierarchy



Lambda Expressions

Anonymous functions as expressions

Assignment Expressions

Assign and evaluate in place

Operators

Tools for building complex expressions

Expression

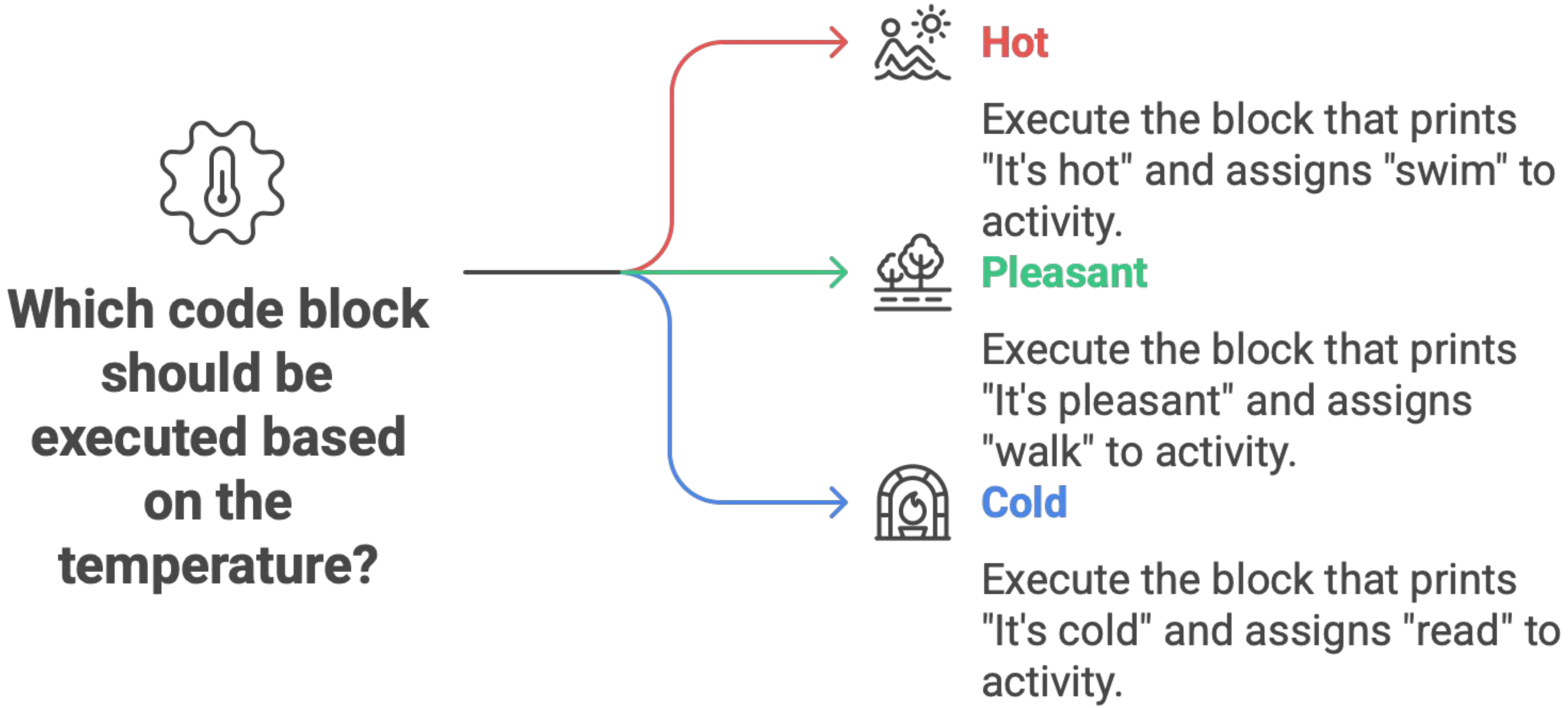
Core code unit evaluating to a value

Statements and Control Flow

Understanding program flow from simple to conditional execution

Executes statements in written order

Executes code block based on condition



Which code block should be executed based on the temperature?

Hot

Execute the block that prints "It's hot" and assigns "swim" to activity.

Pleasant

Execute the block that prints "It's pleasant" and assigns "walk" to activity.

Cold

Execute the block that prints "It's cold" and assigns "read" to activity.

Summary & Look Ahead

Iteration with Loops

Using 'while' and 'for' loops for data processing

Conditional Sequencing

Implementing control flow with 'if' statements

Walrus Operator

Learning the assignment expression

Expressions & Statements

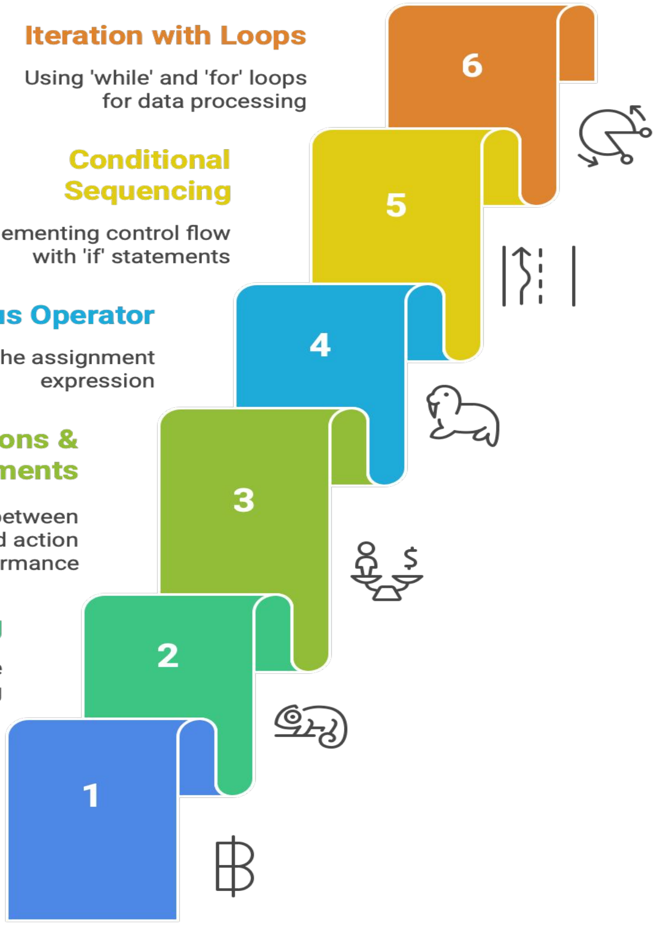
Differentiating between value evaluation and action performance

Dynamic Typing

Grasping Python's flexible variable referencing

Formal Grammars

Understanding how BNF describes valid Python programs



Thank You!