Introduction to Python™

Date: October 8th, 2025

Event: Fall 2025 HPC Training, 9:00 am

Instructor: Oleg N. Starovoytov

Email: olegs@lsu.edu

These training notes have been developed and refined over multiple years to support learners in understanding and applying Python programming. This 3rd edition summarizes key concepts, examples, and practices for educational use.

www.python.org

Disclaimer

The information, examples, or code provided are for educational and informational purposes only. I make no representations or warranties of any kind, express or implied, about the completeness, accuracy, reliability, or suitability of the content. Any reliance you place on such information is strictly at your own risk. I am not liable for any loss or damage, direct or indirect, arising from the use or misuse of this content.

The content of this file is systematically organized into sections to help readers learn and understand the Python programming language more effectively. Each section is divided into Description, Note, Syntax, and Examples. This structure is designed to make the material clear, practical, and easy to follow.

Description

The Description section introduces a specific concept in Python, such as an operator, function, module, or class. It provides an overview of its purpose, behavior, and common use cases. This section helps students build a better understanding and become more familiar with the concept.

Note

The Note section provides additional information that may be helpful but is not essential to the main topic. It can include related concepts, tips, warnings, or reminders, offering useful insights or advice to support your understanding of the material.

Syntax

The Syntax section shows the correct way to write a concept in Python. It focuses on the structure and format, including the order of elements, symbols used, and required components. This sectio ✦ ssential for learning to correctly apply and understand the concept in

programming.

Examples

The Examples section provides practical code snippets that demonstrate how a concept is applied in real scenarios. This section helps reinforce understanding by showing the concept in action.

Introduction

Python™ is a high-level, general-purpose programming language that emphasizes readability and simplicity, making it one of the most widely used languages in the world today. It was first released in 1991 by Guido van Rossum and has since become popular for its clear syntax, extensive libraries, and flexibility. Python is commonly used for tasks such as web development, data analysis, scientific computing, artificial intelligence, machine learning, and automation. Because it is interpreted and cross-platform, Python can be run on all major operating systems, including Linux®, macOS®, and Windows®, as well as on mobile and cloud environments, making it highly versatile.

These training notes, now in their 3rd edition, summarize the essential concepts and practices needed to understand and use Python effectively. The material covers the basics of programming in Python, the use of different modes such as interactive and script execution, as well as working with Jupyter notebooks for cell-based coding. By walking through examples and explanations, the notes provide learners with a foundation to apply Python in various domains, whether for simple scripting tasks or advanced data-driven applications. This edition improves upon previous versions by refining explanations and ensuring coverage of modern practices in Python.

Python Installation

To install Python on any operating system, the best place to start is the official **Python** website at www.python.org , where you can download the latest stable version for Windows, macOS, or Linux. Detailed installation guides, documentation, and tutorials are also available there. For users who prefer managing environments and packages easily, tools such as **Anaconda** (www.anaconda.com) or **Miniconda** (www.docs.conda.io) provide ready-to-use Python distributions with many scientific and data analysis libraries preinstalled. Developers who use Linux can install Python directly from their system's **package manager** (e.g., apt, dnf, or yum), and macOS users can use **Homebrew** (https://brew.sh) for quick setup. For writing and testing Python code **Jupyter Notebook** (https://jupyter.org/try) are among the most popular environments.

Python Version

In this guide, we will be using Python 3.12.11 (main, Jun 4 2025, 08:56:18) [GCC 11.4.0]. All examples and syntax are compatible with **Python 3** and may not work as expected in **Python 2**. It is recommended to run the code in an environment where Python 3.x is installed, such as Google Colab, Jupyter Notebook, or local **Python 3** setup.

Python 2 reached end-of-life in January 2020 and is no longer officially supported. This means that no security updates or bug fixes are provided, many modern libraries no longer support **Python 2**, and code written for **Python 3** may not run correctly in **Python 2**. For these reasons, it is strongly suggested to use Python3.x for all new projects and exercises in this guide.

```
import sys
print(sys.version)
```

```
!python3 --version
```

Python Compiler

When you check your Python version using sys.version, you may see **GCC**, **Clang**, or **MSVC** listed. This indicates which C compiler was used to build the Python interpreter. For running Python code, the choice of compiler usually does not affect usage, but it matters if you are building Python from source or compiling C extensions.

GCC (GNU) refers to the GNU Compiler Collection version. GCC is an open-source compiler suite developed by the GNU Project and supports multiple programming languages, including C, C++, Fortran, Go, and more. Python itself is mostly written in C, so it needs a C compiler to be built from source. GCC is the compiler used to build this Python interpreter, and it affects performance, compatibility with extensions, and how Python runs on your system. When you see Python compiled with GCC, it usually means it was built on **Linux**.

Clang is an open-source compiler for C, C++, Objective-C, and Objective-C++ programming languages. It is part of the LLVM (Low-Level Virtual Machine) project and serves as the front-end compiler that translates your source code into machine code. Clang is known for fast

compilation and clear error messages.When you see Python compiled with Clang, it usually means it was built on **macOS**.

MSVC (Microsoft C/C++) is Microsoft's proprietary C and C++ compiler. It is the standard compiler used to build Python on Windows, including the official CPython distribution. On Windows, Python is typically compiled with MSVC. MSVC compiler ensures compatibility with Windows system libraries and C extensions. When you see Python compiled with MSC, it usually means it was built on **Windows**.

Environmental Variables

PYTHONSTARTUP points to a Python script that is executed automatically every time the interactive Python interpreter is launched. This file is typically used to preload functions, import frequently used modules, or configure interpreter settings for convenience. For example, developers may include common imports like os or sys, or define helper functions to streamline their interactive workflow. The variable only applies to interactive sessions, not to scripts executed directly. It is useful for advanced users who customize their shell. Leave it unset.

PYTHONHOME is an environment variable that defines the location of the Python installation and its standard libraries. It can be used to isolate Python environments or redirect the interpreter to use a different library tree than the system default. This is especially important when embedding Python in applications or managing multiple Python versions. If not explicitly set, Python infers its home from the location of the interpreter binary. Leave it unset.

PYTHONPATH is an environment variable that specifies additional directories for Python to search when importing modules and packages. By default, Python looks in the standard library and the site-packages directory, but adding paths to PYTHONPATH allows you to include custom or project-specific modules without installing them globally. This is commonly used in development environments to make local code accessible during testing or to override default module locations. Leave it unset unless you really need extra module directories.

Set up **PYTHONPATH**: 1. you have custom modules not installed in site-packages; 2. you're developing multiple related projects in different directories; 3. you need to supplement the default module search path.

Syntax

```
export PYTHONPATH=/path/to/custom/modules:$PYTHONPATH
```

Python Interpreter

A **Python interpreter** is a program that reads and executes Python code. It translates the code you write in Python into instructions that your computer's processor can understand and run.

Python is an interpreted language, which means the interpreter reads and runs your code one statement at a time, instead of compiling the entire program into machine code first. The interpreter handles translating Python's high-level syntax into low-level instructions for your computer.

There are three modes that can be used for programming and executing the scripts: 1. **Interactive**, 2. **Script-based** , and 3. **Cell-based modes**:

1. **Interactive mode**: Type python3 in the terminal or Python shell, get the Python prompt (>>>), and type commands in the Python prompt. Here is an example.

```
[username@hostname ~$ python3
Python 3.13.4 (main, Jun 10 2025, 16:30:17) [Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> print ("Hello, World!")
Hello, World!
>>>
```

2. **Script-based mode**: You save code in a file (myScript.py) and run it. Python executes a whole file containing code and saved with .py extension.

```
[username@hostname ~$ python3 myScript.py
Hello, World!
username@hostname ~$
```

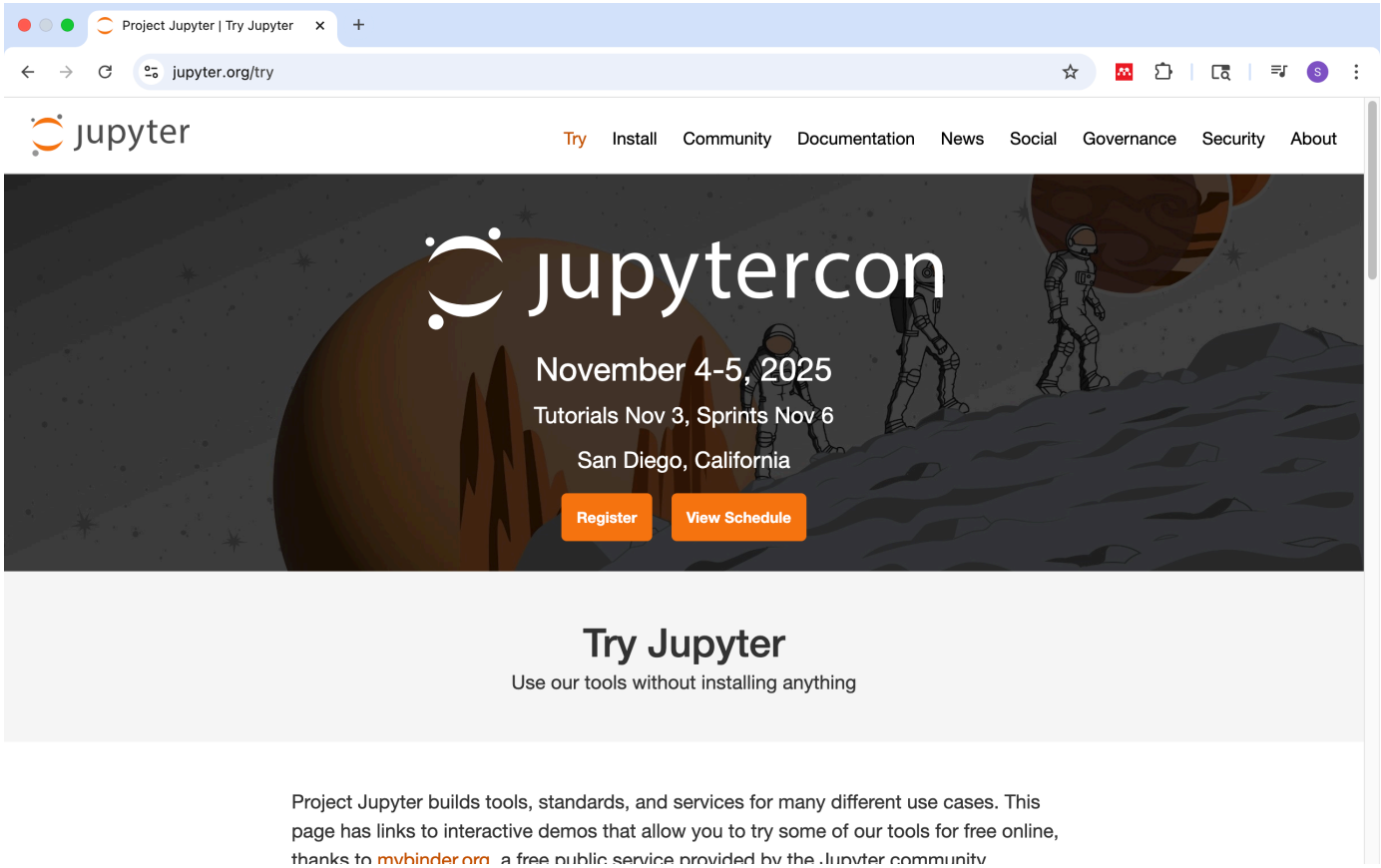
3. **Cell-based mode**: In Jupyter Notebook, code is orginized into cells, which you can execute independently. Each cell behaves like an interactive session, but the notebook keeps a persistent state across cells.

```
print("Hello, World!")
```

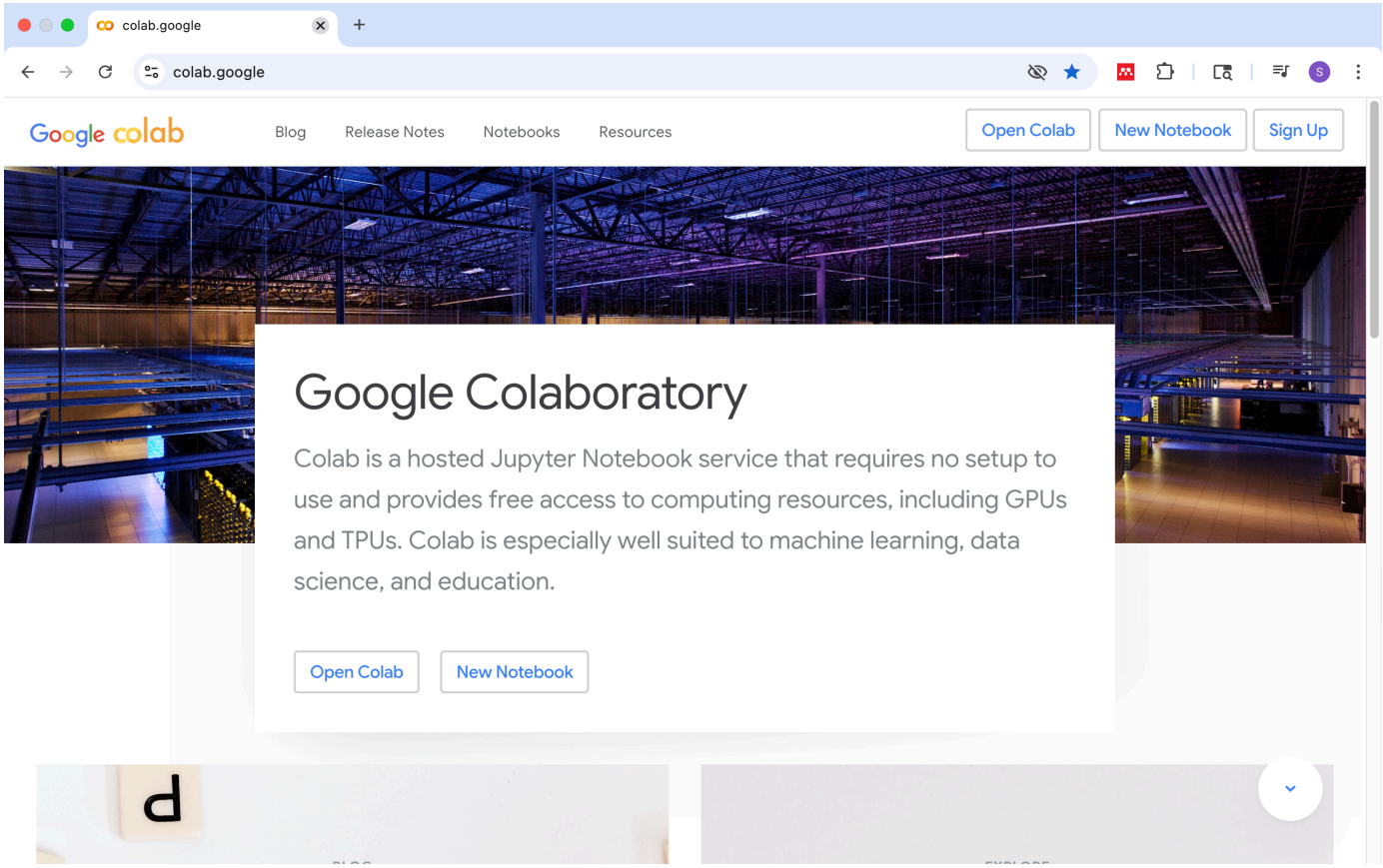
Online Python interpreter

---

1. jupyter.org/try



2. colab.google



Outline

Conventions

Throughout this material, certain conventions are followed to ensure clarity and consistency. Code examples are presented in monospaced font, while commands to be executed in a terminal are prefixed with a prompt symbol, indicating the bash shell environment (e.g., $). Notes, tips, and warnings are **highlighted** where appropriate to draw attention to critical information.

We will use specific variables for clarity and consistency. The variables *a* and *b* will be used to demonstrate bitwise operations, as these typically involve integer values at the binary level. For all other examples - including arithmetic, comparison, logical, assignment, and string operations - we will use variables following the Camel Case convention, such as *myVariable*. Simple names like *x*, *y*, and *z* may also be used for clarity. This approach keeps examples organized and makes it easier to distinguish between general-purpose variables and those used for low-level bitwise operations.

Section 1: Python programming basics

1.1 Print function

Description

The print() function in Python is used to display information on the screen. It can output text, numbers, variables, or the results of expressions. This makes it especially useful for checking values while debugging, monitoring program flow, or presenting final results to the user. Since it is one of the most frequently used functions in Python, getting comfortable with print() is an essential first step in learning the language.

Syntax

```
print(arguments)
```

Examples

```
print("Hello, World!")
```

1.2 Comments

Description

Comments are an essential part of programming, as they make code easier to read and maintain by explaining its purpose and logic. In Python, single-line comments start with the # symbol, and everything after it is ignored by the interpreter. For longer explanations, you can use triple single quotes (' ' ') or triple double quotes (" " "), which are technically string literals but are often used as multi-line comments or docstrings (documentation strings) inside functions, classes, and modules.

Note

“Lorem ipsum dolor sit amet...” is not random Latin but a scrambled section of a passage from Cicero’s writings in 45 BC. Its use became widespread in the 1960s with the rise of typesetting software and has since become a standard placeholder text. It is primarily used in publishing, graphic design, and web development to focus on layout and visual design elements rather than readable content. Known as “dummy text,” it helps give an impression of how the final text will look without distracting readers with meaningful words.

"*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.*"

Syntax

```
     # Lorem ipsum dolor sit amet ...

   ''' Lorem ipsum dolor sit amet ... '''

 " " " Lorem ipsum dolor sit amet ... " " "
```

Examples

```
# This is my first comment

print("Hello, World!")
```

```
# Using triple quotes to make a multiple line comment

''' This is a multiple
    line
    comment '''

print("Hello, World!")
```

```
# Using triple double quotes to make a multiple line comment

""" This is a multiple
    line
    comment """

print("Hello, World!")
```

Note

Professional programmers often include a header comment at the top of each source code file to provide essential information about the program. A typical header may include the author's name, creation date, a brief description of the code, version or update history, license details, contact information, and any other relevant notes. This practice improves readability, helps with collaboration, and ensures proper documentation of the source file.

Examples

```
# Function: sort(myFirstArray)
# Author:   Firstname Lastname (somename@address.edu)
# Date:     12/10/1815
# Updated:  06/10/1835
# Description: The sort function sorts an array of integers.
# Parameters:  Array (list of integers)
# Return:      Sorted array
# Example of usage:
#     myFirstArray = [23, 20, 45, 14, 2]
#     sortedArray  = [2, 14, 20, 23, 45]

def sort(myFirstArray):
    return sorted(myFirstArray)

# Example usage
myFirstArray = [23, 20, 45, 14, 2]
sortedArray = sort(myFirstArray)
print(sortedArray)
```

1.3 Variables

Description

Variables are used to store data values that can be referenced and manipulated later in a program. In Python, a variable is created when you assign it a value using the assignment operator (=), and unlike many other programming languages, you do not need to explicitly declare its type. Python automatically determines the type based on the value assigned. A variable's name serves as a reference to an object in memory, and each object can belong to different data types, holding corresponding values.

Syntax

```
myVariable = value
```

Examples

```
# Illegal declaration of variables

2025year      = "Future"    # Cannot start with a number
student-name  = "Alice"     # Cannot contain a dash (-)
total marks   = 95          # Cannot contain spaces
$salary       = 50000       # Cannot contain special characters
class         = "Physics"   # Cannot use reserved Python keywords
```

```
# Legal declaration of variables

year2025      = "Future"    # String
student_name  = "Alice"     # String with underscore
total_marks   = 95          # Integer
salary_usd    = 50000.75    # Float
is_graduated  = True        # Boolean
_value        = None        # NoneType
```

```
# Usual declaration of variables

a = 1
b = 'Hello, World'
c = -100000000000
d = 0.345
e = True
f = 5.0E-51
g = 1_000_000_000
A_65_ASCII = "A"
```

Note

Naming conventions are widely used in programming to make variables, functions, and classes easier to read and maintain. In Python, the most common style for variables and functions is snake_case (e.g., total_marks), while class names are typically written in PascalCase (e.g., StudentRecord). You may also encounter camelCase (e.g., studentName) in some contexts, but it is less common in Python. Following consistent naming conventions improves code readability and collaboration. Classes should be named using PascalCase (also called UpperCamelCase), where each word begins with a capital letter and no underscores are used—for example, MyClass, StudentRecord, or DataProcessor.

Syntax

Camel Case - each word starts with a capital letter except the first one.

```
myVariableName
```

Pascal Case - each word starts with a capital letter.

```
MyVariableName
```

Snake Case - each word is connected by underscore.

```
my_variable_name
```

Examples

```
# Declaration of a variable using Camel Case

myCamelCase = "Camel Case"

print(myCamelCase)
```

```
# Declaration of a variable using Pascal Case

MyPascalCase = "Pascal Case"

print(MyPascalCase)
```

```
# Declaration of a variable using Snake Case

my_snake_case = "Snake Case"

print(my_snake_case)
```

Note

Variables in Python are created and initialized at the moment a value is first assigned to them. Unlike many other programming languages, you do not need to declare the variable type in advance, as Python automatically determines the type based on the assigned value. There are multiple ways to assign values to variables, including direct assignment, multiple assignments in one line, and assigning the same value to multiple variables.

Examples

```
# Assign multiple values to multiple variables in one line

x, y, z = 23.4, 35.6, 45.1

print(x)
print(y)
print(z)
```

```
# Assign multiple values of multiple data types to multiple variables in one line

x, atomName, stateID = 23.4, "Carbon", True

print(x)
print(atomName)
print(stateID)
```

```
# Assign one value to multiple variables

x = y = z = 2.25

print(x)
print(y)
print(z)
```

```
# Using unpacking method to assign values to variables

coordinates = [2.35, 6.37, -8.94]
x, y, z = coordinates

print(x)
print(y)
print(z)
```

```
# When unpacking a list, you can use _ to ignore certan values.

coordinates = [2.35, 6.37, -8.94]
x, _, z = coordinates

print(x)
print(z)
```

```
# Using * to ignore multiple values when unpacking
coordinates = [2.35, 6.37, 7.12, 9.45, -8.94]
x, *_, z = coordinates

print(x)
print(z)
```

Note

In Python, variables are references to objects, and while you can reassign them or modify their contents, there are cases where you may need to remove them or free up memory. To delete a variable or remove its reference to an object, you can use the del statement, which makes the variable name undefined until it is reassigned.

Syntax

```
del myVariable
```

Examples

```
# Deleting a variable using the del command

myVariable = 65

print (myVariable)
del myVariable
```

Note

Global and local variables:

Global variables are defined outside of functions and can be accessed from anywhere in the code, whereas local variables are created within a function and can only be accessed inside that function's scope. Understanding the interaction between global and local variables is important for managing data and avoiding conflicts; more details on this topic can be found in the Function section.

Note

In Python, naming conventions with underscores convey special meanings.

A single leading underscore (e.g., _myVariable) indicates that a variable or method is intended for internal use and should be treated as private, although this is only a convention and not enforced.

A double leading underscore (e.g., _ _myVariable) triggers name mangling, where the interpreter changes the variable's name internally to reduce the chance of accidental overriding in subclasses.

A single trailing underscore (e.g., myVariable_) is often used to avoid conflicts with reserved Python keywords; for example, class_ can be used instead of the reserved keyword class.

1.4 Data types

Description

Data types define the kind of values a variable can hold, and the datatype of a variable is determined at runtime. Python provides many built-in data types, including numeric types (*int*, *float*, *complex*) for numbers, sequence types (*str*, *list*, *tuple*) for ordered collections, the

mapping type (*dict*) for key-value pairs, and set types (*set*, *frozenset*) for unordered collections of unique elements. It also includes the Boolean type (*bool*) and NoneType to represent the absence of a value. Additionally, Python allows you to create custom data types using classes. Understanding data types is essential for writing correct and efficient code, since different operations and methods apply to different types.

Note

The five **primitive data types** in Python are *int*, *float*, *str*, *bool*, and *NoneType*, which serve as the foundation for all data structures and expressions. All other types, such as *lists*, *tuples*, *dictionaries*, *sets*, and *files*, are considered **non-primitive**. Python defines all of its data types, including primitive ones, using classes. Primitive types are immutable, meaning their values cannot be changed after creation, which makes them reliable building blocks for managing data in programs.

Syntax

- Numeric Types: *int, float, complex*
- Text Type: *str*
- Boolean Type: *bool*
- Sequence Types: *list, tuple, range*
- Mapping Type: *dict*
- Set Types: *set, frozenset*
- Binary Types: *bytes, bytearray, memoryview*
- None Type: *None*

Note

One can use the *type()* function to determine the data type of a variable.

Syntax

```
type(myVariable)
```

Examples

```
# Using type function: type()

# Numeric types
numInt       = 23                # int
numFloat     = 0.00036           # float
numComplex   = 10 + 5j           # complex
isActive     = True              # bool

# Sequence types
textString   = "Hello, World!"               # str
numbersList  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # list
namesTuple   = ("John", "Mike", "Desbey")     # tuple

# Mapping type
personDict   = {"Name": "John", "Age": 41, "Occupation": "Doctor"}  # dict

# Set types
fruitsSet    = {"apple", "banana", "cherry"}                      # set
frozenFruits = frozenset({"apple", "banana", "cherry"})           # frozenset

# Binary types
dataBytes    = b"Hello"             # bytes
dataByteArr  = bytearray(5)         # bytearray
dataMemView  = memoryview(bytes(5)) # memoryview

# Other types
emptyValue   = None       # NoneType
rangeObject  = range(12)  # range
```

```
# Print variable types in a loop
variables = [
    numInt, numFloat, numComplex, isActive,
    textString, numbersList, namesTuple, personDict,
    fruitsSet, frozenFruits, dataBytes, dataByteArr,
    dataMemView, emptyValue, rangeObject
]

for var in variables:
    print(type(var))
```

1.5 Literals

Description

Literals are fixed values that are directly represented as objects in Python. The five main types of literals are numeric, Boolean, string, collection, and special literals. When a literal is assigned to a variable, the variable becomes a reference to the corresponding object in memory. Since everything in Python is an object, every literal is an instance of a class, and variables simply point to these objects.

Note

Numeric literals: *integer, float, complex*

Boolean literals: *True, False*

String literals: *text surrounded by single, double, or triple quotes*

Collection literals: *List, Tuple, Dict, Set*

Special literal: *None*

Syntax

```
myVariable = literal
```

Examples

```
# Example of literals in Python

# Numeric literals
numLiteralInt     = 0               # Integer literal
numLiteralBin     = 0b10100         # Binary literal
numLiteralDec     = 50              # Decimal literal
numLiteralFloat   = 234.8           # Float literal
numLiteralComp    = 2 + 3j          # Complex literal

# Boolean literal
numLiteralBool    = True            # Boolean literal

# String literals
strLiteralSingle  = 'Hello'         # Single-quoted string
strLiteralDouble  = "World"         # Double-quoted string
strLiteralMulti   = """This is
a multi-line string"""               # Triple-quoted string

# Collection literals
listLiteral       = [1, 2, 3, 4]                    # List literal
tupleLiteral      = (10, 20, 30)                    # Tuple literal
dictLiteral       = {"name": "Alice", "age": 25}    # Dictionary literal
setLiteral        = {"apple", "banana", "cherry"}   # Set literal
frozensetLiteral  = frozenset({1, 2, 3})            # Frozenset literal

# Special literal
noneLiteral       = None            # NoneType literal

# Print types to show that literals create objects
```

```
variables = [
    numLiteralInt, numLiteralBin, numLiteralDec, numLiteralFloat, numLiteralComp,
    numLiteralBool,
    strLiteralSingle, strLiteralDouble, strLiteralMulti,
    listLiteral, tupleLiteral, dictLiteral, setLiteral, frozensetLiteral,
    noneLiteral
]

for var in variables:
    print(type(var))
```

Note

Literals in Python directly represent objects, and assigning a literal to a variable simply creates a reference to that object in memory. Everything in Python is an object, and each literal value corresponds to an instance of a class. Each object is an instance of a class, and every value in Python is also an instance of a class. Classes define the behavior and properties of their instances, so literals ultimately follow the rules and methods of the classes they belong to.

1.6 Numbers

Description

The primary numeric data types in Python are integers, floating-point numbers, and complex numbers. Integers are whole numbers without a decimal point or fractional part and can be positive, negative, or zero. They have arbitrary precision, meaning they can grow as large as memory allows. Floating-point numbers represent real numbers with decimals and fractional parts, following the IEEE 754 double-precision standard, with a precision of approximately 15–17 decimal digits. Complex numbers have a real and an imaginary part, where the imaginary unit is written as j. Numeric types are commonly used in mathematical, engineering, and scientific applications.

Syntax

```
myVariable = numeric datatype
```

Examples

```
# Integer
myIntNumber = 1

# Float
myFloatNumber = 1.0

# Complex
myComplexNumber = 1.0j
```

Binary, octal, and hexadecimal numbers

Description

In Python, you can work with *binary*, *octal*, and *hexadecimal* numbers using different prefixes and built-in functions. Here's how you can represent and work with these numeric bases.

Examples

Binary (base 2) numbers are represented using two digits: 0 and 1. In Python, binary literals are written with the prefix **0b** or **0B**, followed by the binary digits. The *bin()* function is used to convert an integer to its binary representation as a string.

```
# Binary numbers use the bin() function for conversion
# and the prefix 0b (or 0B) to represent binary literals.
```

```
binaryValue  = 0b101   # Binary literal for decimal 5
decimalValue = 5       # Decimal representation

print(binaryValue)
print(decimalValue)
```

Octal (base 8) numbers are represented using eight digits: 0–7. In Python, octal literals are written with the prefix **0o** (zero followed by a lowercase **o**). The *oct()* function is used to convert an integer to its octal representation as a string.

```
# Octal numbers use the oct() function for conversion
# and the prefix 0o (zero followed by a lowercase o) to represent octal literals.

octalValue   = 0o5   # Octal literal for decimal 5
decimalValue = 5     # Decimal representation

print(octalValue)
print(decimalValue)
```

Hexadecimal (base 16) numbers are represented using ten digits (0–9) and six letters (A–F), where A represents 10, B represents 11, C represents 12, D represents 13, E represents 14, and F represents 15. In Python, hexadecimal literals are written with the prefix **0x** (zero followed by a lowercase **x**). The *hex()* function is used to convert an integer to its hexadecimal representation as a string.

```
# Hexadecimal numbers use the hex() function for conversion
# and the prefix 0x (zero followed by a lowercase x) to represent hexadecimal literals.

hexValue     = 0x5   # Hexadecimal literal for decimal 5
decimalValue = 5     # Decimal representation

print(hexValue)
print(decimalValue)
```

1.7 Casting

Description

Casting refers to the process of converting a value from one data type to another. This is useful when performing operations that require a specific data type. Type conversion in Python can occur either implicitly (automatic conversion by the interpreter) or explicitly (manual conversion using built-in functions). Explicit conversions are typically done using functions such as *int()*, *float()*, or *str()*, which act as constructors for their respective data types.

Syntax

```
myVariable = function(VariableName)
```

Examples

```
# Implicit type conversion (type coercion)
# Python automatically converts an integer to a float during arithmetic with a float.

integerValue = 1
print(type(integerValue))

resultValue = integerValue + 0.25
print(type(resultValue))
```

```
# Explicit type conversion (type casting)
# Converting an integer to string and float using built-in functions.

integerValue = 1
stringValue  = str(integerValue)
floatValue   = float(integerValue)
```

```
print(type(integerValue))
print(type(stringValue))
print(type(floatValue))
```

```
# Using casting to create complex numbers

x, y = 1, 2

complexNumber1 = complex(x, y)    # 1 + 2j
complexNumber2 = complex(3)       # 3 + 0j
complexNumber3 = complex(4, 5)    # 4 + 5j

print("Complex number 1:", complexNumber1)
print("Complex number 2:", complexNumber2)
print("Complex number 3:", complexNumber3)
```

1.8 Boolean

Description

Booleans in Python can be either **True** or **False**. The value **True** corresponds to **1**, and **False** corresponds to **0**. Note that Python is case-sensitive, so **True** is not the same as **TRUE** or **true**, and **False** is not the same as **FALSE** or **false**. Booleans are commonly used in conditionals, comparisons, and logical operations.

Syntax

```
myVariable = Boolean value
```

Examples

```
# Boolean values in Python
# Booleans can be either True or False

print("Boolean True value:", True)
print("Boolean False value:", False)
```

```
# Casting Boolean values to integers
# True converts to 1, and False converts to 0

print("Integer value of True:", int(True))
print("Integer value of False:", int(False))
```

```
# Boolean arithmetic with numbers
# In Python, True is treated as 1 and False as 0 in numeric operations

print("True + 1 =", True + 1)
print("False + 1 =", False + 1)
```

```
# Comparing Boolean values with numbers
# True is equal to 1 and False is equal to 0 in Python

print("Is True equal to 1?", True == 1)
print("Is False equal to 0?", False == 0)
```

1.9 Strings

Description

Strings are an immutable collection of alphanumeric characters enclosed in either single ('...') or double ("...") quotes. String literals cannot be changed after they are created. A double-quoted string may contain single quotes without escaping them. Characters within a string can be accessed using indexing, with indices starting at 0. If the index is out of range, Python will raise an IndexError. To access a range of characters, you can use slicing, specifying the start and end indices enclosed in square brackets, as shown below.

Syntax

```
myString = "Lorem ..."
```

Examples

```
# Assigning a string to a variable using single quotes

myString = 'This is my first string'
print(myString)
```

```
# Assigning a string to a variable using double quotes

myString = "This is my second string"
print(myString)
```

```
# Assigning a string to a variable using single and double quotes

myString = "This's my third string"  # Double quotes allow single quotes inside
print(myString)
```

```
# Assign string to a variable using single quotes

myString = 'That's my fourth string'
```

```
# Using escape sequences in strings
# The backslash (\) allows inclusion of special characters like single quotes

myString = 'That\'s my fourth string'
print(myString)
```

```
# Printing a backslash character
# Use a double backslash \\ to represent a single backslash in a string

backslashChar = "\\"
print(backslashChar)
```

```
# Assigning a multi-line string to a variable using triple single quotes

multiLineString = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''

print(multiLineString)
```

```
# Assigning a multi-line string to a variable using triple double quotes

multiLineString = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""

print(multiLineString)
```

String indexing

Description

Indexing is an important concept in Python. Each character of a string has a unique index number. To access any character of a string, you use its corresponding index. String indexing works similarly to lists and tuples, but unlike strings, sets are not indexed because they are unordered. The index of a string starts at **0**, so the first character has index **0**, the second character has index **1**, and so on.

Syntax

```
    myString[index]
```

Examples

```
# Strings elements can be accessed by their index

# The indexes
# myTestString   =   [1,  2,  3,  4,  5,  6,  7,  8,  9, ..., 26]
# --------------------------
# index              0,  1,  2,  3,  4,  5,  6,  7,  8, ..., n-1
# negative index    -n,                                  ...,  -1
# myTestString[index]

myTestString = "Lorem ipsum dolor sit amet"

print("Character at index 4:", myTestString[4])           # 'm'
print("Character at index 5:", myTestString[5])           # ' '
print("Character at index 6:", myTestString[6])           # 'i'
print("Character at index 18:", myTestString[18])         # 's'
print("Character at index -1:", myTestString[-1])         # 't'
print("Character at index -26:", myTestString[-26])       # 'L'

# Length of the string
print("Length of the string:", len(myTestString)) # 26
```

String slicing

Description

The **slicing technique** allows you to access a **range of characters** in a string. The range is specified using a **start index** and an **end index** inside square brackets. The start index is **inclusive**, meaning the character at this index is **included** in the slice, while the end index is **exclusive**, meaning the character at this index is **not included**. In other words, a slice includes characters from the start index up to, but not including, the end index.

Syntax

```
    myString[start index: end index: step]
```

Examples

```
# Strings elements can be accessed by index

# The indexes
# myTestString   =   [1,  2,  3,  4,  5,  6,  7,  8,  9, ..., 26]
# --------------------------
# index              0,  1,  2,  3,  4,  5,  6,  7,  8, ..., n-1
# negative index    -n,                                  ...,  -1
# myTestString[start index: end index]

sampleText = "Lorem ipsum dolor sit amet"

print("Characters from start to index 4:", sampleText[:4])       # 'Lore'
```

```
print("Characters from index 0 to 5:", sampleText[0:5])          # 'Lorem'
print("Characters from index 0 to 6:", sampleText[0:6])          # 'Lorem '
print("Characters from index 0 to 18:", sampleText[0:18])        # 'Lorem ipsum dolor '
print("Characters from index -12 to -1:", sampleText[-12:-1])    # 'or sit ame'
print("Characters from index -26 to -25:", sampleText[-26:-25])  # 'L'

# Length of the string
print("Length of the string:", len(sampleText))                  # 26
```

String methods

Description

Strings are one of the most commonly used data types in Python. To help you work with them efficiently, Python provides many built-in methods that allow you to perform a variety of operations on string values. Below are some widely used string methods with examples.

Syntax

```
    myString.method()
```

Examples

```
# Converting a string to uppercase using the upper() method
# Returns a new string with all characters in uppercase

myString = "Hello, World!"
print("Original string:", myString)
print("Uppercase string:", myString.upper())
```

```
# Converting a string to lowercase using the lower() method
# Returns a new string with all characters in lowercase

myString = "Hello, World!"
print("Original string:", myString)
print("Lowercase string:", myString.lower())
```

```
# Removing leading and trailing whitespace using the strip() method

myString = "  Hello, World!  "
print("Original string:", repr(myString))      # Shows whitespace clearly
print("Stripped string:", myString.strip())    # Whitespace removed
```

```
# Replacing a substring using the replace() method

myString = "Hello, World!"
print("Original string:", myString)
print("Replaced string:", myString.replace("Hello", "Good morning"))
```

```
# Splitting a string into a list using the split() method
# By default, split() uses whitespace as the separator

myString = "Good morning, World!"
print("Original string:", myString)
print("Split string:", myString.split())
```

```
# Using replace() to remove a character and split() to create a list of words

myString = "Good morning, World!"

# Remove the comma
cleanString = myString.replace(',', '')

# Remove leading/trailing whitespace and split into a list of words
wordList = cleanString.strip().split()
```

```
print("Original string:", myString)
print("Cleaned string:", cleanString.strip())
print("List of words:", wordList)
```

String operations

Description

Several basic operations can be performed on strings in Python, including concatenation (+) to combine strings, repetition (*) to repeat a string multiple times, indexing ([]) to access individual characters, slicing ([:]) to extract a range of characters, and membership (in) to check if a substring exists within a string. These operations allow you to combine, repeat, access, and manipulate strings efficiently.

Syntax

Concatenation: result = string1 + string2

Repetition: result = string * number_of_times

Indexing: character = string[index]

Slicing: substring = string[start_index:end_index:step]

Membership check: exists = substring in string

Examples

```
# Concatenating two strings using the + operator

firstString = "Hello"
secondString = "World!"

# Combine the strings
combinedString = firstString + secondString
print("Concatenated string:", combinedString)
```

```
# Repeating strings using the * operator

star = "*"
smile = ")"

print(star * 2)               # '**'
print(star * 4)               # '****'
print(star * 6 + " " + smile) # '****** )'
```

```
# Accessing individual characters in a string using indexing

sampleText = "Lorem ipsum dolor sit amet"

print("Character at index 4:",   sampleText[4])   # 'm'
print("Character at index 5:",   sampleText[5])   # ' '
print("Character at index 6:",   sampleText[6])   # 'i'
print("Character at index 18:", sampleText[18])   # 's'
print("Character at index -1:", sampleText[-1])   # 't'
```

```
# Accessing a range of characters in a string using slicing

sampleText = "Lorem ipsum dolor sit amet"

print("Characters from start to index 4:",   sampleText[:4])   # 'Lore'
print("Characters from index 0 to 5:",      sampleText[0:5])   # 'Lorem'
print("Characters from index 0 to 6:",      sampleText[0:6])   # 'Lorem '
print("Characters from index 0 to 18:",    sampleText[0:18])   # 'Lorem ipsum dolor '
```

```
# Checking if a substring exists within a string using the 'in' operator

sampleText = "Lorem ipsum dolor sit amet"

print("'Lorem' in sampleText:", "Lorem" in sampleText)  # True
print("'lorem' in sampleText:", "lorem" in sampleText)  # False
```

```
# Strings are immutable in Python
# You cannot change individual characters of a string directly

myString = "Hello, World!"

try:
    myString[0] = "G"  # Attempting to change the first character
except TypeError as e:
    print("Error:", e)

# Correct way to modify a string is to create a new string
newString = "G" + myString[1:]
print("Original string:", myString)
print("Modified string:", newString)
```

Section 2: Operators

2.1 Arithmetic operators

Description

Arithmetic operators in Python are used to perform basic mathematical operations. The primary operators include addition (+), subtraction (-), multiplication (*), exponentiation (**), division (/), modulus (%), and floor division (//). Note that the division operator (/) always returns a float, even if both operands are integers. The modulus operator (%) returns the remainder of a division, while floor division (//) returns the largest integer less than or equal to the result. These operators work with integers, floats, and even complex numbers, allowing for flexible numerical computations in Python.

Syntax

- Addition: x + y
- Subtraction: x - y
- Multiplication: x * y
- Exponentiation: x ** y
- Division: x / y
- Floor division: x // y
- Modulus: x % y

Examples

```
# Addition operator (+): adds two numbers

firstNumber = 1
```

```
secondNumber = 2

sumResult = firstNumber + secondNumber

print("Sum of the numbers:", sumResult)  # 3
```

```
# Subtraction operator (-): subtracts one number from another

firstNumber = 2
secondNumber = 1

subResult = firstNumber - secondNumber

print("Subtraction result:", subResult)  # 1
```

```
# Multiplication operator (*): multiplies two numbers

firstNumber = 2
secondNumber = 3

multResult = firstNumber * secondNumber

print("Multiplication result:", multResult)  # 6
```

```
# Exponentiation operator (**): raises a number to the power of another

baseNumber = 2
exponent = 8

result = baseNumber ** exponent

print("Result of exponentiation:", result)  # 256
```

Note

The division operator / always returns a floating-point number, even if the division is for two integers. The operator works with both integers and floating-point numbers and can be used in mathematical expressions or combined with variables. Division by zero is not allowed and will raise a ZeroDivisionError.

```
# Division operator (/): always returns a float

numeratorNumber = 8
denominatorNumber = 2

divisionResult = numeratorNumber / denominatorNumber

print("Division result:", divisionResult)  # 4.0
```

Note

Floor division // always rounds down to the nearest integer, not toward zero.

```
# Floor division operator (//): returns the largest integer less than or equal to the result
# Example: quotient = dividend // divisor
#          quotient = 10 // 3
#          quotient = 3

dividendNumber = 10
divisorNumber = 3

floorDivisionResult = dividendNumber // divisorNumber

print("Floor division result:", floorDivisionResult)  # 3
```

```
# Floor division operator (//): returns the largest integer less than or equal to the result
# Formula for negative dividend: quotient = -ceil(-dividend / divisor)
# Example: quotient = -ceil(-(-10) / 3)
```

```
#          quotient = -ceil(10 / 3)
#          quotient = -4

dividendNumber = -10
divisorNumber = 3

floorDivisionResult = dividendNumber // divisorNumber

print("Floor division result:", floorDivisionResult)  # -4
```

```
Floor division result: -4
```

```
# Floor division operator (//): returns the largest integer less than or equal to the result
# Example: quotient = dividend // divisor
#          quotient = 10 // -3
#          quotient = -4

dividendNumber = 10
divisorNumber = -3

floorDivisionResult = dividendNumber // divisorNumber

print("Floor division result:", floorDivisionResult)  # -4
```

```
Floor division result: -4
```

Note

Modulus operator % returns the remainder, which takes the sign of the divisor. The sign of the remainder always matches the divisor, not the dividend.

```
# Modulus operator (%): returns the remainder of a division
# Formula: remainder = dividend - (divisor * quotient)
# Example: remainder = 8 - (2 * 4) = 0

dividendNumber = 8
divisorNumber = 2

modulusResult = dividendNumber % divisorNumber

print("Modulus result:", modulusResult)  # 0
```

```
# Modulus operator (%): returns the remainder of a division
# Formula: remainder = dividend - (divisor * (dividend // divisor))
# Example: remainder = -10 - (3 * (-10 // 3))
#          remainder = -10 - (3 * -4)
#          remainder = 2

dividendNumber = -10
divisorNumber = 3

modulusResult = dividendNumber % divisorNumber

print("Modulus result:", modulusResult)  # 2
```

```
# Modulus operator (%): returns the remainder of a division
# Formula: remainder = dividend - (divisor * (dividend // divisor))
# Example: remainder = -10 - (-3 * (-10 // -3))
#          remainder = -10 - (-3 * 3)
#          remainder = -1

dividendNumber = -10
divisorNumber = -3

modulusResult = dividendNumber % divisorNumber

print("Modulus result:", modulusResult)  # -1
```

Note

A complex number is a number that has two parts: a real part and an imaginary part, written in the form a + bj, where a is the real part, b is the imaginary part, and j represents the square root of -1. Complex numbers are used in mathematics, physics, and engineering to represent quantities that cannot be expressed with real numbers alone, such as electrical currents, wave functions, or oscillations. Python provides built-in support for complex numbers, allowing you to create, manipulate, and perform arithmetic operations on them directly using standard operators (+, -, *, /) or the complex() constructor.

```
# Performing arithmetic operations on complex numbers
# Let z1 = a + bj and z2 = c + dj
# Addition:       z1 + z2 = (a + c) + (b + d)j
# Subtraction:    z1 - z2 = (a - c) + (b - d)j
# Multiplication: z1 * z2 = (a*c - b*d) + (a*d + b*c)j
# Division:       z1 / z2 = [(a*c + b*d) / (c*c + d*d)] + [(b*c - a*d)/(c*c + d*d)]j

z1 = 3 + 4j
z2 = 1 + 5j

print("Addition result:       ", z1 + z2)  # 4 + 9j
print("Subtraction result:    ", z1 - z2)  # 2 - 1j
print("Multiplication result: ", z1 * z2)  # -17 + 19j
print("Division result:       ", z1 / z2)  # 0.8235294117647058 - 0.4117647058823529j
```

Assignment operators

Description

Assignment operators in Python are used to assign values to variables and can also perform mathematical operations while assigning. The basic assignment operator **=** assigns the value on the right-hand side to the variable on the left-hand side. Python also supports compound assignment operators, which combine assignment with arithmetic or bitwise operations, including add and assign **(+=)**, subtract and assign **(-=)**, multiply and assign **(*=)**, divide and assign **(/=)**, floor divide and assign **(//=)**, modulus and assign **(%=)**, exponentiate and assign **(**=)**, and the assignment expression **(:=)**, also called the walrus operator. These operators allow for concise, readable code, especially when updating a variable based on its current value.

Syntax

- Assignment: x = y
- Add and assign: x += y
- Subtract and assign: x -= y
- Multiply and assign: x *= y
- Divide and assign: x /= y
- Floor divide and assign: x //= y
- Modulus and assign: x %= y
- Exponentiate and assign: x **= y

Examples

```
# Assignment operator (=): assigns the value of one variable to another

originalValue = 5
assignedValue = originalValue

print("Assigned value:", assignedValue)  # 5
```

```
# Addition assignment operator (+=): adds a value to the variable and assigns the result

currentValue = 5
currentValue += 30  # Equivalent to: currentValue = currentValue + 30

print("Updated value:", currentValue)  # 35
```

```
# Multiplication assignment operator (*=): multiplies the variable by a value and assigns the result
```

```
currentValue = 3
currentValue *= 30  # Equivalent to: currentValue = currentValue * 30

print("Updated value:", currentValue)  # 90
```

```
# Division assignment operator (/=): divides the variable by a value and assigns the result

currentValue = 6
currentValue /= 4.0  # Equivalent to: currentValue = currentValue / 4.0

print("Updated value:", currentValue)  # 1.5
```

```
# Floor division assignment operator (//=): divides the variable by a value using floor division and assigns the qu

currentValue = 13
currentValue //= 4  # Equivalent to: currentValue = currentValue // 4

print("Updated value after floor division:", currentValue)  # 3
```

```
# Modulus assignment operator (%=): divides the variable by a value and assigns the remainder

currentValue = 6
currentValue %= 4  # Equivalent to: currentValue = currentValue % 4

print("Updated value after modulus:", currentValue)  # 2
```

```
# Exponentiation assignment operator (**=): raises the variable to the power of a value and assigns the result

baseNumber = 2
baseNumber **= 8  # Equivalent to: baseNumber = baseNumber ** 8

print("Updated value after exponentiation:", baseNumber)  # 256
```

Note

The assignment expression operator **(:=)**, also called the walrus operator because it resembles a walrus with tusks, allows you to assign a value to a variable as part of an expression. It is particularly useful in **if** and **while** statements, as well as in **list comprehensions**, **filtering**, and **mapping** operations. Introduced in Python 3.8, the walrus operator enables assignment and return of a value in a single expression, reducing redundancy and improving readability. For examples in **control flow**, see Section 4.

Syntax

```
myVariable := expression
```

Examples

```
# Standard input and output

userName = input("Enter your name: ")
print(f"Hello, {userName}")
```

```
# Using the walrus operator (:=) to assign and check input in a single expression

if (userName := input("Enter your name: ")) != "":
    print(f"Hello, {userName}")
```

2.2 Comparison operators

Description

Comparison operators are used to **compare two values** and return a **Boolean result**: either **True** or **False**. The main operators include: equal to **(==)**, not equal to **(!=)**, greater than **(>)**, less than **(<)**, **greater than or equal to **(>=)**, and less than or equal to **(<=)**. These operators are commonly used in conditional statements such as **if** and **while**, as well as in logical expressions, to control the flow of a program. They can be applied to numbers, strings, and other data types to **evaluate relationships between values or expressions efficiently**.

Syntax

- Equal: x == y
- Not equal: x != y
- Greater than: x > y
- Less than: x < y
- Greater than or equal to: x >= y
- Less than or equal to: x <= y

Examples:

```
# Equality operator (==): checks if two values are equal and returns True or False

firstNumber = 10.0
secondNumber = 10.0

print("Are the numbers equal?", firstNumber == secondNumber)  # True
```

```
# Not equal operator (!=): checks if two values are not equal and returns True or False

firstNumber = 10.0
secondNumber = 9.0

print("Are the numbers not equal?", firstNumber != secondNumber)  # True
```

```
# Greater than operator (>): checks if the first value is greater than the second

firstNumber = 10.0
secondNumber = 9.0

print("Is the first number greater than the second?", firstNumber > secondNumber)  # True
```

```
# Less than operator (<): checks if the first value is less than the second

firstNumber = 10.0
secondNumber = 9.0

print("Is the second number less than the first?", secondNumber < firstNumber)  # True
```

```
# Greater than or equal to operator (>=): checks if the first value is greater than or equal to the second

firstNumber = 10.0
secondNumber = 10.0

print("Is the second number greater than or equal to the first?", secondNumber >= firstNumber)  # True
```

```
# Less than or equal to operator (<=): checks if the first value is less than or equal to the second

firstNumber = 10.0
secondNumber = 10.0

print("Is the second number less than or equal to the first?", secondNumber <= firstNumber)  # True
```

Note

String comparisons in Python are lexicographical, meaning that the characters of the strings are compared one by one using their ASCII values. For example, when comparing "ABCDa" and "ABCDb", Python first compares the first four characters (A, B, C, D) which are equal,

so the result of the comparison depends on the fifth character, where a has an ASCII value of 97 and b has an ASCII value of 98. As a result, "ABCDa" is considered less than "ABCDb", and equality, greater-than, or less-than decisions are determined by the first character pair that differs.

```
# String comparison (case-sensitive) using ASCII values
# ASCII Table: A=65, B=66, C=67, D=68, a=97, b=98, c=99, d=100

firstChar = "A"
secondChar = "A"

print("Are the characters equal?", firstChar == secondChar)  # True
```

```
# String comparison (case-sensitive) using ASCII values
# ASCII Table: A=65, B=66, C=67, D=68, a=97, b=98, c=99, d=100

firstString = "ABCDa"
secondString = "ABCDb"

print("Are the strings equal?", firstString == secondString)   # False
print("Is the first string greater than the second?", firstString > secondString)  # False
print("Is the first string less than the second?", firstString < secondString)    # True
```

Note

Complex numbers are a **built-in numeric type** in Python. They can be compared for **equality** using == or !=, but **ordering comparisons** such as <, >, <=, or >= **are not allowed** and will raise a **TypeError**. For example, (3 + 5j) == (3 + 5j) returns **True**, while (3 + 4j) < (5 + 2j) raises an **error**. This restriction exists because complex numbers do not have a natural ordering in mathematics, and Python enforces this limitation. If ordering is required—for instance, when sorting a list of complex numbers—you must provide a **custom key**, often using the **magnitude** (via abs()) or the **real** or **imaginary parts**.

The abs() function returns the **magnitude** (or modulus) of a complex number, calculated as the square root of the sum of the squares of its real and imaginary parts:

abs(a+bj) = sqrt(a^2 + b^2)

This allows comparison or sorting based on the size of the complex number rather than its value in the complex plane.

Examples

```
# Complex numbers can be compared for equality using == or !=

firstComplex  = 3 + 5j
secondComplex = 3 + 5j

print("Are the complex numbers equal?", firstComplex == secondComplex)  # True
```

```
# Complex numbers can be compared using their magnitudes (absolute values)

firstComplex  = 3 + 5j
secondComplex = 3 + 4j

print("Is the magnitude of the first complex number less than the second?",
      abs(firstComplex) < abs(secondComplex))  # False
```

```
# Complex numbers can not be compared using >, <, <=, >= operators.

firstComplex  = 3 + 5j
secondComplex = 2 + 3j

print(myFirstComplex > mySecondComplex)
```

2.3 Logical operators

Description

Logical operators in Python are used to perform Boolean logic and combine multiple conditional expressions, returning **True** or **False** based on their relationship. The main logical operators are **and**, **or**, and **not**. The **and** operator returns **True** only if both conditions are true, **or** returns **True** if at least one condition is true, and **not** inverts the result, returning **True** when the condition is false. These operators are commonly used in conditional statements to construct complex decision-making logic in programs.

Note

The **or** operator in Python uses short-circuit evaluation, meaning that if the first condition is **True**, Python does not evaluate the second condition. It immediately returns **True** because the overall expression has already satisfied the requirement for an or operation.

Syntax

```
condition1 and condition2

condition1 or  condition2

not condition
```

Examples

```
# Logical operator: and
# Returns True only if both conditions are True

firstNumber  = 10.0
secondNumber = 10.0
thirdNumber  = 20.0

print("Are both conditions true?", firstNumber < thirdNumber and firstNumber == secondNumber)  # True
```

```
# Logical operator: or
# Returns True if at least one condition is True

firstNumber  = 10.0
secondNumber = 10.0
thirdNumber  = 20.0

print("Is at least one condition true?", firstNumber > thirdNumber or firstNumber == secondNumber)  # True
```

```
# Logical operator: not
# Inverts the Boolean value of the condition

firstNumber  = 10.0
secondNumber = 20.0

print("Inverted condition result:", not firstNumber > secondNumber)  # True
```

2.4 Bitwise operators

Description

Bitwise operators in Python are used to perform **Boolean logic operations on individual bits** of integers. They operate at the **binary level**, similar to logical operators but on the bit representation of numbers. Common bitwise operators include **AND** (&), **OR** (|), **XOR** (^), and **NOT** (~). These operators are useful for **evaluating and manipulating data at the bit level**, making them important in low-level programming tasks such as setting flags, applying masks, or interfacing with hardware. Note that **both operands must be integers** for bitwise operations to work.

Syntax

- AND: a & b

- OR: a | b
- XOR: a ^ b
- NOT: ~a
- Left shift: a << n
- Right shift: a >> n

Examples

```
# Bitwise operator: AND (&) – performs a logical conjunction at the bit level
# 6 = 00000110
# 3 = 00000011
#----------------------
# 2 = 00000010

print(6&3)
```

```
# Bitwise operator: OR (|) – performs a logical disjunction at the bit level
# 6 = 00000110
# 3 = 00000011
#----------------------
# 7 = 00000111

print(6|3)
```

```
# Bitwise operator: NOT (~) – performs a logical negation at the bit level
#  3 = 00000011
#---------------------------
# -4 = 11111100

print(~3)
```

```
# Bitwise operator: XOR (^) – performs exclusive OR at the bit level
# 6 = 00000110
# 3 = 00000011
#---------------------------
# 5 = 00000101

print(6^3)
```

```
# Bitwise operator: << – left shift (shifts bits to the left, inserting zeroes on the right)
# 3  = 00000011
#-------------------------
# 12 = 00001100

print(3<<2)
```

```
# Bitwise operator: >> – right shift (shifts bits to the right)
# 8 = 00001000
#-----------------------
# 2 = 00000010

print(8>>2)
```

Note

When working with numbers greater than 255, you'll need more than one byte to represent them. A single byte can hold values from 0 to 255 (in decimal). Anything larger than 255 requires additional bytes.

For example:

0–255 ⟶ 1 byte

256–65,535 ⟶ 2 bytes

65,536–16,777,215 ⟶ 3 bytes

and so on.

So, the number 500 requires 2 bytes to be stored. Its binary representation is: 00000001 11110100 ⟶ which is 9 bits, crossing the 1-byte limit.

Assignment operators

Description

Bitwise assignment operators combine a bitwise operation with assignment, allowing you to modify a variable by applying a bitwise operation directly to it. These operators include **bitwise AND and assignment (&=)**, **bitwise OR and assignment (|=)**, **bitwise XOR and assignment (^=)**, **left shift and assignment ** **(<<=)**, and **right shift and assignment** **(>>=)**. They are commonly used in **low-level programming tasks**, such as manipulating binary data, managing flags, or interfacing with hardware, where direct bit-level control is required.

Syntax

- Bitwise AND and assignment: a &= b
- Bitwise OR and assignment: a |= b
- Bitwise XOR and assignment: a ^= b
- Bitwise right shift and assignment: a >>= b
- Bitwise left shift and assignment: a <<= b

Examples:

```
# Bitwise AND assignment operator: &=
# Performs a bitwise AND and assigns the result to the variable
# Example: 6 & 3 = 2

bitValue = 6

bitValue &= 3  # Equivalent to: bitValue = bitValue & 3

print("Bitwise AND assignment result:", bitValue)  # 2
```

```
# Bitwise OR assignment operator: |=
# Performs a bitwise OR and assigns the result to the variable
# Example: 6 | 3 = 7

bitValue = 6

bitValue |= 3  # Equivalent to: bitValue = bitValue | 3

print("Bitwise OR assignment result:", bitValue)  # 7
```

```
# Bitwise XOR assignment operator: ^=
# Performs a bitwise XOR and assigns the result to the variable
# Example: 6 ^ 3 = 5

bitValue = 6

bitValue ^= 3  # Equivalent to: bitValue = bitValue ^ 3

print("Bitwise XOR assignment result:", bitValue)  # 5
```

```
# Bitwise left shift assignment operator: <<=
# Shifts the bits of the variable to the left and assigns the result
# Example: 3 << 2 = 12

bitValue = 3

bitValue <<= 2  # Equivalent to: bitValue = bitValue << 2

print("Bitwise left shift assignment result:", bitValue)  # 12
```

```
# Bitwise right shift assignment operator: >>=
# Shifts the bits of the variable to the right and assigns the result
# Example: 8 >> 2 = 2

bitValue = 8

bitValue >>= 2  # Equivalent to: bitValue = bitValue >> 2

print("Bitwise right shift assignment result:", bitValue)  # 2
```

2.5 Identity operators

Description

Identity operators are used to compare the memory location, or identity, of two objects. Unlike **comparison operators**, which check if the values of two objects are equal, **identity operators** determine whether two variables reference the exact same object in memory. Python provides two identity operators: **is** and **is not**. The **is** operator returns **True** if both variables point to the same object, while **is not** returns **True** if they point to different objects. It's important to note that two variables can have equal values but still be distinct objects. Identity operators should be used when the actual object identity matters, not just equality of values.

Syntax

```
x is y
x is not y
```

Examples:

```
# Identity operator: is
# Checks whether two variables reference the same object in memory

stringA = "Hello World!"
stringB = "Hello World!"

stringC = stringA

print(stringA is stringB)  # False or True depending on interning
print(stringA is stringC)  # True, same object

# Print memory addresses to see object identity
print("Memory addresses:", id(stringA), id(stringB))
print("Memory addresses:", id(stringA), id(stringC))
```

```
# Identity operator: is not
# Checks whether two variables reference different objects in memory

stringA = "Hello World!"
stringB = "Hello World!"

stringC = stringA

print(stringA is not stringB)  # False or True depending on interning
print(stringA is not stringC)  # False, same object

# Print memory addresses to illustrate object identity
print("Memory addresses:", id(stringA), id(stringB))
print("Memory addresses:", id(stringA), id(stringC))
```

2.6 Membership operators

Description

Membership operators in Python are used to test whether a value exists within a sequence or collection, such as **lists**, **strings**, **tuples**, or **sets**. There are two membership operators: **in** and **not in**. The **in** operator returns **True** if the specified value is present in the collection, while **not in** returns **True** if the value is absent. These operators are commonly used in conditional statements to simplify code, enhance readability, and avoid verbose loops or checks.

Syntax

```
value in sequence

value not in sequence
```

Examples

```
# Membership operator: in – checks if an element exists in a collection

fruits = ["apple", "banana", "cherry", "date"]

# Check if "banana" is in the list
print("banana" in fruits)  # True

# Check if "orange" is in the list
print("orange" in fruits)  # False
```

```
# Membership operator: not in – checks if an element does NOT exist in a collection

fruits = ["apple", "banana", "cherry", "date"]

# Check if "orange" is NOT in the list
print("orange" not in fruits)  # True

# Check if "banana" is NOT in the list
print("banana" not in fruits)  # False
```

2.7 Operator precedence

Description

Operator precedence determines the order in which operators are evaluated in complex expressions. **Operators with higher precedence are executed before those with lower precedence**, **unless parentheses are used to explicitly specify the order.** For example, exponentiation **()**** has higher precedence than multiplication **(*)** and addition **(+)**, which in turn have higher precedence than **comparison** and **logical** operators. Understanding operator precedence is essential for writing clear and predictable expressions and avoiding unexpected results.

Syntax

- Parentheses: ()
- Exponentiation: **
- Unary Operators: +x, -x, ~x
- Arithmetic Operators: *, /, //, %, +, -
- Bitwise Operators: <<, >>, &, ^, |
- Comparison Operators: ==, !=, >, <, >=, <=
- Assignment Operators: =, +=, -=, *=, /=, ...
- Identity Operators: is, is not
- Membership Operators: in, not in
- Logical Operators: not, and, or

Note

In Python, operator precedence determines the order in which expressions are evaluated.

Arithmetic follows PEMDAS:

Parentheses → Exponentiation → Multiply/Divide/Floor/Mod → Add/Subtract.

Unary operators like positive (+) and negative (-) are applied before binary operations. Bitwise operators (&, ^, |) are evaluated after arithmetic but before comparisons. Comparison operators (==, !=, <, >, <=, >=) come next, followed by assignment operators (=, +=, -= etc.). Finally, logical operators are evaluated last, in the order not → and → or, and identity (is, is not) and membership (in, not in) operators are evaluated alongside comparisons.

A simple memory trick to recall this sequence is: "Please Excuse My Dear Aunt Sally & XOR OR Not"

where P → Parentheses, E → Exponentiation, M/D → Multiply/Divide/Floor/Mod, A/S → Add/Subtract, &/^/| → Bitwise AND/XOR/OR, Comparisons → Assignment → Logical NOT/AND/OR.

Section 3: Data Collections

3.1 List

Description

A **list** in Python is a *built-in data type* used to store multiple items in a single variable. Lists are **ordered** and **mutable**, meaning their contents can be changed after creation. They are commonly used to store collections of related data, and their elements can be accessed by **index**, starting from 0. Python provides many built-in methods to **modify and manipulate lists**, including inserting new items, deleting existing ones, and performing operations like **slicing**, **concatenation**, and **multiplication**. Lists are a member of the sequence data type, allowing iteration, membership testing, and other sequence operations.

List declaration

Description

A list is created by placing a sequence of elements inside square brackets [ ], separated by commas. A list can contain elements of any data type, including numbers, strings, or even other lists (nested lists).

Syntax

```
myList = [element 1, element 2, ..., element n]

myList = [element 1]

myList = list()
```

Examples

```
# Creating an empty list
myEmptyList = []
```

```
print("My empty list:", myEmptyList)  # Output: []
print(type(myEmptyList))              # Output: <class 'list'>
```

```
# Creating an empty list using the constructor
myEmptyList = list()

print("My empty list:", myEmptyList)  # Output: []
print(type(myEmptyList))              # Output: <class 'list'>
```

```
# Creating a list of strings
myFishList = ["Red Snapper", "Carp", "Bass"]

print(myFishList)  # Output: ['Red Snapper', 'Carp', 'Bass']
```

```
# Creating a list of integers
myIntList = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(myIntList)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# List with mixed data types
myMixedList  = [10, 11.0, 123, "Python", "A", -234.5, True, False, 1.23E-24]
myNumberList = [1, 2, 3, 4, 5]

# Printing both lists separated by a semicolon
print(myMixedList, myNumberList, sep=" ; ")
```

```
# Creating a list using the list() constructor
myFishList = list(("Carp", "Brass", "Red Snapper"))  # note the round brackets for the tuple

print(myFishList)       # Output: ['Carp', 'Brass', 'Red Snapper']
print(type(myFishList)) # Output: <class 'list'>
```

```
# Original list
myFishList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Creating a copy of the list
myNewList  = list(myFishList)

print(myNewList)            # Output: ['Carp', 'Brass', 'Salmon', 'Red Snapper']
print(myNewList is myFishList)  # Output: False
```

```
# List of fish
myFishList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Get the length of the list
print(len(myFishList))  # Output: 4
```

List indexing

Description

Each element in a Python list has a unique index number, starting from 0 for the first element. You can access any element using its index in square brackets [ ], just like with strings. Positive indices count from the beginning of the list (0 for the first element, 1 for the second, etc.), while negative indices count from the end (-1 for the last element, -2 for the second-last, and so on). This indexing allows you to retrieve or modify specific elements within the list efficiently.

Syntax

```
myList[index]
```

Examples

```
# The indexes
# myFirstList   =   [1,  2,  3,  4,  5, ..., n]
# --------------------------
# index              0,  1,  2,  3,  4, ..., n-1
# negative index    -n,                 ...,  -1

myFirstList = ["Red Snapper", "Carp", "Bass"]

print(myFirstList[0])
print(myFirstList[1])
print(myFirstList[2])
```

List slicing

Description

List slicing allows you to extract a portion of a list by specifying a range of indices using the colon : operator inside square brackets. The slice returns a new list containing elements from the start index up to, but not including, the stop index. You can also provide an optional step value to skip elements. Slicing is a powerful way to access and manipulate sublists without modifying the original list.

A nested list is a list that contains other lists as elements, allowing you to represent more complex data structures such as matrices, tables, or multi-dimensional arrays. Nested lists are useful for organizing related data in a structured and hierarchical way.

Syntax

```
    myList[start index: end index: step]
```

Example

```
# The indexes
# myFirstList   =   [1,  2,  3,  4,  5]
# --------------------------
# index              0,  1,  2,  3,  4
# negative index    -5, -4, -3, -2, -1

myFirstList = [1, 2, 3, 4, 5]

print(myFirstList)
print(myFirstList[:])
print(myFirstList[1:])
print(myFirstList[0:3])
print(myFirstList[3:4])
print(myFirstList[3:5])
print(myFirstList[-4:])
print(myFirstList[-4:-1])
print(myFirstList[-5])
```

```
# Multi-dimensional (nested) list example

myNestedList = [1, 2, 3, [10, 20, 30], 'String']

print(myNestedList)
print(type(myNestedList))
print(len(myNestedList))
```

```
myNestedList = [1, 2, 3, [10, 20, 30], 'String']

# Access the second element of the nested list
print(myNestedList[3][1])  # Output: 20
```

```
# Using the sorted() function, Syntax: sorted(iterable, key = ..., reverse = ...)

myNumberList = [1.2, -0.2, 3.0, 0.6]
```

```
# Sort in ascending order
print(sorted(myNumberList))  # Output: [-0.2, 0.6, 1.2, 3.0]
```

```
# Using the sorted() function, list by the length of each string
myNestedList = ["Lorem", "ipsum", "do", "sit", "amet", "consectetur", "elit"]

print(sorted(myNestedList, key=len))
```

```
# Using the sorted() function with the lambda function, ord(), and max() functions

mySrtList = [ "Lorem", "ipsum", "do", "sit", "amet", "consectetur", "elit"]

print(sorted(mySrtList, key = lambda Chr: ord(max(Chr))))
```

```
# Use the in operator to check if an element exists in a list

myFishList = ["Carp", "Brass", "Red Snapper"]

# Check if "Fly" is in the list
print("Brass" in myFishList)
```

```
# Assigning a list with multiple elements expands the original list.

# Original list containing three elements
myFishList = ["Carp", "Brass", "Red Snapper"]

# Replace a slice of the list
# myFishList[2:3] selects the element at index 2 ("Red Snapper")
# Assigning ["Trout", "Shark"] replaces that slice with two new elements
myFishList[2:3] = ["Trout", "Shark"]

# Print the updated list
print(myFishList)
```

```
['Carp', 'Brass', 'Trout', 'Shark']
```

List methods

Description

There are many **built-in list methods** in Python that allow you to **manipulate and interact with lists** easily. Some of the most commonly used methods include *insert()*, *append()*, *extend()*, *remove()*, *pop()*, *clear()*, *sort()*, and *copy()*. These methods provide convenient ways to **add**, **remove**, **organize**, and **duplicate** elements in a list without writing complex code, making list management more efficient and readable.

Syntax

```
myList.method()
```

Examples

```
# Insert method: insert() - inserts an element into the list

# Original list
myFishingSet = ["Carp", "Brass", "Fly", "Spider"]

# Insert "Spinning" at index 2 (before "Fly")
myFishingSet.insert(2, "Spinning")

# Insert "Casting" at index 3 (before "Fly" after previous insertion)
myFishingSet.insert(3, "Casting")
```

```
# Print the updated list
print(myFishingSet)
```

```
# Append method: append() - adds a single element to the end of the list

# Original list with one element
myFishList = ["Carp"]

# Append new elements to the list one by one
myFishList.append("Brass")
myFishList.append("Red Snapper")
myFishList.append("Shark")

# Print the updated list
print(myFishList)
```

```
# Extend method: extend() - adds multiple elements (from another iterable) to the end of a list

# Original list of fish
myFishList = ["Carp", "Brass", "Red Snapper"]

# Another list containing fishing lures
myLureList = ["Fly", "Spider", "Worm"]

# Extend myFishList by adding all elements from myLureList
myFishList.extend(myLureList)

# Print the updated list
print(myFishList)
```

```
# Remove method: remove() - removes the first occurrence of a specified element from the list

# Original list of fish
myFishList = ["Carp", "Brass", "Red Snapper"]

# Remove the element "Carp" from the list
myFishList.remove("Carp")

# Print the updated list
print(myFishList)
```

```
# Pop method: pop() - removes and returns an element at a specified index
# If no index is specified, it removes and returns the last element in the list

# Original list of fish
myFishList = ["Carp", "Brass", "Red Snapper"]

# Remove the element at index 1 ("Brass")
myFishList.pop(1)

# Print the updated list after removing "Brass"
print(myFishList)
# Output: ['Carp', 'Red Snapper']

# Remove the last element ("Red Snapper") since no index is specified
myFishList.pop()

# Print the updated list after removing the last element
print(myFishList)
```

```
# Clear method: clear() - removes all elements from a list

# Original list containing three lures
myLureList = ["Fly", "Spider", "Worm"]

# Remove all elements from the list, leaving it empty
myLureList.clear()

# Print the now-empty list
print(myLureList)
```

```
# Sort method: sort() - sorts all the elements of a list (case sensitive)

# Original list of fish names
myFishList = ["Carp", "Brass", "Red Snapper", "Shark", "Arowana", "Catfish"]

# Sort the list in ascending (alphabetical) order
# Note: sort() modifies the original list in place
# It is case-sensitive — uppercase letters come before lowercase in ASCII order
myFishList.sort()

# Print the sorted list
print(myFishList)
```

```
# Sort method: sort() - sorts all the elements of a list (case sensitive)

# Original list of fish names
myFishList = ["Carp", "Brass", "Red Snapper", "Shark", "Arowana", "Catfish"]

# Sort the list in descending (reverse alphabetical) order
# The 'reverse=True' argument sorts elements from Z to A
# This operation modifies the original list directly
myFishList.sort(reverse=True)

# Print the sorted list
print(myFishList)
```

```
# Sort method: sort() - sorts all the elements of a list (case insensitive)

# Original list of fish names with mixed uppercase and lowercase letters
myFishList = ["Carp", "Brass", "red Snapper", "shark", "Arowana", "catfish"]

# Sort the list alphabetically in a case-insensitive manner
# The key=str.lower ensures all items are compared in lowercase during sorting
# Syntax: sort(key = ..., reverse = ...)
myFishList.sort(key=str.lower)

# Print the sorted list
print(myFishList)
```

```
# Copy method: copy() - returns a shallow copy of the list

# Original list of fish names
myFishList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Create a copy of the list using the copy() method
myNewList = myFishList.copy()

# Print the copied list
print(myNewList)  # Output: ['Carp', 'Brass', 'Salmon', 'Red Snapper']

# Modify the original list
myFishList[0] = "Pike"

# Print the copied list again to show that it is unaffected by the change
print(myNewList)
```

```
# Assigning to the same memory address of a list

# Original list of fish names
myFishList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Assign myFishList to myNewList (both point to the same memory location)
myNewList = myFishList

# Print the new list (initially identical)
print(myNewList)  # Output: ['Carp', 'Brass', 'Salmon', 'Red Snapper']

# Modify the original list
myFishList[0] = "Pike"

# Print the new list again to show that it was also affected
print(myNewList)
```

List operations

Description

Basic list operations include common tasks that allow us to work with lists effectively. These operations include concatenation, repetition, length determination, iteration, and membership testing. They form the foundation for working with lists and are widely used in everyday Python programming. To delete an entire list, we can use the built-in del statement. This removes the list object from memory, making it no longer accessible within the program.

Syntax

- Concatenation: newList = listOne + listTwo
- Repetition: repeatedList = listOne * n
- Length: listLength = len(listOne)
- Iteration: for listItem in listOne: print(listItem)
- Membership testing: if element in listOne: print("Element found")
- Delete entire list: del listOne

Examples

```
# Join two lists to create a new list using the concatenation operator (+)

# Define two separate lists
myFishList   = ["Carp", "Brass", "Salmon", "Red Snapper"]
myInsectList = ["Fly", "Beetle", "Ant", "Dragonfly"]

# Concatenate both lists into a new list
myFishingSet = myFishList + myInsectList

# Print the combined list
print(myFishingSet)
```

```
# Multiply one list twice using the repetition operator (*)

# Define the original list
myFishList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Repeat the list twice
myFishingSet = myFishList * 2

# Print the repeated list
print(myFishingSet)
```

```
# Find the length of a list using the len() function

# Define a list
```

```
myList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Print the number of elements in the list
print(len(myList))
```

```
# Iteration: Loop through each element in a list

# Define a list
myList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Loop through each element and print it
for item in myList:
    print(item)
```

```
# Membership testing: Check if a value exists in a list

# Define a list
myList = ["Carp", "Brass", "Salmon", "Red Snapper"]

# Check if 'Carp' is in the list
print('Carp' in myList)
```

```
# Delete an entire list using the del statement
# After this, trying to access myFishList will raise a NameError
# print(myFishList)  # Uncommenting this line will cause an error

# Define a list
myFishList = ["Carp", "Brass", "Red Snapper"]

# Delete the list from memory
del myFishList
```

3.2 Tuple

Description

Tuples are used to store multiple values in a single variable. Once a tuple is created, **it cannot be changed**, meaning it is **immutable**. Tuples are **ordered**, so the sequence of elements remains as defined, and they can contain **duplicate values**. A tuple is enclosed in **parentheses ()** and is **indexed** like strings and lists, starting at 0. **Slicing** is also supported, allowing access to a range of elements. Tuples can store elements of **different data types** and are often used to group related data that should not be modified. Their immutability makes them **memory-efficient** and safe to use as **dictionary keys**.

Tuple declaration

Description

A tuple can be declared by placing a sequence of values inside parentheses (), separated by commas. Tuples can be created in various ways, including using the tuple() constructor. If you need a tuple with only one element, you must include a trailing comma after the element to distinguish it from a regular value. Although tuples are immutable and their elements cannot be changed, you can delete an entire tuple using the del statement.

Syntax

```
myTuple = (element 1, element 2, ..., element n)

myTuple = (element 1,)

myTuple = element 1, element 2, ..., element n

myTuple = tuple(iterable)
```

Examples

```
# Creating an empty tuple using parentheses

# Define an empty tuple
myEmptyTuple = ()

# Print the empty tuple and its type
print("My empty tuple:", myEmptyTuple)
print(type(myEmptyTuple))
```

```
My empty tuple: ()
<class 'tuple'>
```

```
# Creating an empty tuple using the tuple() constructor

# Define an empty tuple
myEmptyTuple = tuple()

# Print the empty tuple and its type
print("My empty tuple:", myEmptyTuple)
print(type(myEmptyTuple))
```

```
# Creating a tuple with a single element

# Define a tuple with one element (comma is required)
myTuple = (1, )

# Print the type to confirm it is a tuple
print(type(myTuple))
```

```
# Creating a tuple from a list using the tuple() constructor

# Define a list
myList = [1, 2, 3, 4, 5]

# Convert the list into a tuple
myTuple = tuple(myList)

# Print the type to confirm it is a tuple
print(type(myTuple))
```

```
# Creating a tuple with multiple elements, including duplicates

# Define a tuple without using parentheses (optional)
myTuple = 1, 2, 3, 3

# Print the type to confirm it is a tuple
print(type(myTuple))  # Output: <class 'tuple'>

# Print the tuple
print(myTuple)
```

```
# Creating a tuple from a string

# Convert a string into a tuple
myTuple = tuple("abc")

# Print the resulting tuple
print(myTuple)
```

```
# Creating a tuple with multiple string values

# Define a tuple with several strings (tuple packing)
myFirstTuple = ("Carp", "Salmon", "Shark")

# Print the tuple
print(myFirstTuple)
```

```
('Carp', 'Salmon', 'Shark')
```

```
# Unpacking a tuple into individual variables

# Define a tuple (tuple packing)
myFirstTuple = ("Carp", "Salmon", "Shark")

# Unpack the tuple into separate variables
(myFirstElement, mySecondElement, myThirdElement) = myFirstTuple

# Print each variable
print(myFirstElement)
print(mySecondElement)
print(myThirdElement)
```

```
# Creating a tuple with one item

myOneItemTuple = ("Salmon",)  # Note the trailing comma
print(type(myOneItemTuple))

# THE FOLLOWING IS NOT A TUPLE, THIS IS A STRING!
myNotTuple = ("Salmon")
print(type(myNotTuple))
```

```
# Creating a tuple using the range() function

# Convert a range of numbers into a tuple
myRangeTuple = tuple(range(1, 10))  # Numbers from 1 to 9

# Print the resulting tuple
print(myRangeTuple)
```

```
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
# Tuple constructor: tuple() - creates a tuple from an iterable

# Create a tuple using the tuple() constructor
mySecondTuple = tuple(("Salmon", "Tuna"))

# Print the resulting tuple
print(mySecondTuple)
```

```
('Salmon', 'Tuna')
```

```
# Deleting a tuple using the del statement

# Create a tuple
mySecondTuple = tuple(("Salmon", "Tuna"))

# Delete the tuple from memory
# After this, trying to access mySecondTuple will raise a NameError
# print(mySecondTuple)  # Uncommenting this line will cause an error
del mySecondTuple
```

Tuple indexing

Description

Tuple indexing allows you to access individual elements using their positions, starting from 0 for the first element. Tuples are indexed similarly to strings and lists, and you can use square brackets [] with the index number to obtain a specific value. Since tuples are immutable, their elements can be accessed by indexing but cannot be changed once assigned. Indexing is a fundamental operation when working with tuples, especially for referencing or retrieving their values.

Syntax

```
myTuple[index]
```

Examples

```
# Access a tuple with the index

# myTuple       =   [1,  2,  3,  4,  5, ..., n]
# --------------------------
# index              0,  1,  2,  3,  4, ..., n-1
# negative index    -n,                 ...,  -1

# Tuple indexing with a fish tuple

myFishTuple = ("Carp", "Salmon", "Tuna")

# Access the third element (index 2)
print("This fish is found in many rivers and oceans:", myFishTuple[2])
```

Tuple slicing

Description

Tuple slicing allows you to extract a portion of a tuple by specifying a range of indices using the colon (:) operator. The basic syntax returns a new tuple containing elements from the start index up to, but not including, the stop index. You can also include an optional step value to skip elements. Slicing does not modify the original tuple and works the same way as for lists, making it a convenient and powerful method to access elements.

Syntax

```
myTuple[start index: end index: step]
```

Examples

```
# Slice tuple with the index numbers
# Slicing does not modify the original tuple; it returns a new tuple

# myTuple       =   [1,  2,  3,  4,  5, ..., n]
# --------------------------
# index              0,  1,  2,  3,  4, ..., n-1
# negative index    -n,                 ...,  -1

# Tuple slicing: Extract portions of a tuple

# Define a tuple with mixed data types
myMixedTuple = (10, 11.0, 123, "Python", "A", -234.5, True, False, 1.23E-24)

# Slice the first 5 elements (index 0 to 4)
print(myMixedTuple[:5])

# Slice elements from the third-last to the second-last (index -3 to -1)
print(myMixedTuple[-3:-1])
```

```
(10, 11.0, 123, 'Python', 'A')
(True, False)
```

Tuple methods

Description

Tuple methods are limited due to the **immutable nature** of tuples; their contents **cannot be changed or manipulated**. Only two **built-in methods** are available: *count()* and *index()*. These methods are useful for obtaining information from a tuple, such as checking for

duplicates or locating specific elements. All other list-like methods — including *sort()*, *append()*, *insert()*, *copy()*, *extend()*, *pop()*, *clear()*, *reverse()*, and *remove()* — do not work on tuples because they attempt to modify the tuple, which is immutable. However, some methods that do not modify the tuple can still be used to query or access its elements.

Syntax

```
myTuple.method(value)
```

Examples

```
# Count method: count() - returns the number of occurrences of an element in a tuple

# Define a tuple with repeated elements
myTestTuple = (1, 2, 3, 4, 5, 5, 5, 5, 5, 5, 6)

# Count how many times the value 5 appears in the tuple
print(myTestTuple.count(5))
```

```
6
```

```
# Index method: index() - returns the index of the first occurrence of the specified element

# Define a tuple
myTestTuple = (1, 2, 3, 4, 5, 6, 7, 8, 9)

# Get the index of the first occurrence of 4
print(myTestTuple.index(4))  # Output: 3
```

```
3
```

```
# Demonstrating tuple immutability: attempting to modify a tuple

# Define a tuple
myTestTuple = (1, 2, 3, 4, 5, 6, 7, 8, 9)

# The following operations are not allowed on tuples and will raise errors if uncommented:
# myTestTuple.append(10)    # AttributeError: 'tuple' object has no attribute 'append'
# myTestTuple.insert(0, 10) # AttributeError: 'tuple' object has no attribute 'insert'
# del myTestTuple[1]         # TypeError: 'tuple' object doesn't support item deletion
# myTestTuple[0] = 10        # TypeError: 'tuple' object does not support item assignment

# Print the length of the tuple (allowed)
print(len(myTestTuple))
```

```
# Using the sorted() function on a tuple

# Define a tuple with numeric elements
myTestTuple = (12, 2, 1, 4, -5, 6, -7, 8, 9)

# The sorted() function returns a sorted list from the tuple
sortedTuple = sorted(myTestTuple)

# Print the sorted list
print(sortedTuple)
```

Tuple operations

Description

Tuple operators allow you to perform basic operations on tuples. Common operations include **concatenation** (+), **repetition** (*), **indexing** ([index]), and **membership testing** (in, not in). The **del** statement can be used to delete an entire tuple. After deletion, any attempt to access the tuple will raise a NameError because the tuple no longer exists in memory.

Syntax

- Concatenation: newTuple = firstTuple + secondTuple
- Repetition: repeatedTuple = firstTuple * n
- Indexing: elementValue = firstTuple[elementIndex]
- Membership testing: if elementValue in firstTuple: print("Element found")
- Deletion: del firstTuple

Examples

```
# Join two tuples using concatenation

myFishTuple = ("Pike", "Bass", "Carp")
myMammalTuple = ("Blue Whale", "Orca", "Dolphin")

# Concatenate the tuples
myAnimalTuple = myFishTuple + myMammalTuple

# Print the resulting tuple
print(myAnimalTuple)
```

```
# Repeat a tuple using the * operator

myFishTuple = ("Pike", "Bass", "Carp")

# Multiply the tuple to repeat its elements
myAnimalTuple = myFishTuple * 2

# Print the resulting tuple
print(myAnimalTuple)
```

```
# Using membership operators with tuples

myFishTuple = ("Pike", "Bass", "Carp")

# Check if an element exists in the tuple
print("Carp" in myFishTuple)
print("Salmon" not in myFishTuple)
```

```
 # Deleting an entire tuple using del

myFishTuple = ("Pike", "Bass", "Carp")

# Remove the tuple from memory
# After deletion, accessing myFishTuple will raise a NameError
# print(myFishTuple)  # Uncommenting this line will cause an error

del myFishTuple
```

3.3 Dictionary

Description

A **dictionary** is a **built-in** Python data type used to store data in **key:value pairs**, enclosed in curly braces {}. Each key is separated from its value by a colon :, and pairs are separated by commas. Dictionaries are **ordered** (starting from Python 3.7) and **mutable**, allowing their items to be added, modified, or removed. Keys must be **unique and immutable** (such as strings, numbers, or tuples), while values can be of **any data type** and may be repeated. Dictionary items are **accessed via their keys** rather than **numeric indices**, making dictionaries highly efficient for **lookup, insertion, and updates**. They are ideal for representing labeled or structured data.

Dictionary declaration

Description

A dictionary can be declared by creating a set of key:value pairs enclosed within curly braces {}. Each key is followed by a colon : and paired with its corresponding value, and multiple key-value pairs are separated by commas. Alternatively, dictionaries can be created using the dict() constructor. This flexible structure allows for fast lookups, efficient updates, and easy management of structured data, making dictionaries a powerful tool for organizing information in Python.

Syntax

```
myDictionary = {key 1: value, key 2: value, ..., key n: value}

myDictionary = {key: value}

myDictionary = dict()
```

Examples

```
# Declaring an empty dictionary

# Create an empty dictionary using curly braces
myEmptyDict = {}

# Print the empty dictionary
print("My empty dictionary :", myEmptyDict)
```

```
# Declaring an empty dictionary using the dict() constructor

# Create an empty dictionary
myEmptyDict = dict()

# Print the empty dictionary
print("My empty dictionary :", myEmptyDict)
```

```
# Adding new items to the dictionary using [key] = value pairs

myFishDict = {}

myFishDict["Species"] = "Salmon"
myFishDict["Habitat"] = "River"
myFishDict["LengthCm"] = 75

# Print the dictionary
print(myFishDict)
```

```
# Adding a new item to the dictionary with multiple values using one key

myFishDict = {}

myFishDict["Species"] = "Salmon"
myFishDict["Habitat"] = "River"
myFishDict["LengthsCm"] = 50, 60, 70, 80  # multiple values as a tuple

# Print the dictionary
print(myFishDict)
```

```
# Declaring a dictionary with keys that have multiple values

myFishDict = {
    "Species": ("Salmon", "Trout"),
    "Habitat": "River",
    "LengthsCm": (50, 60, 70)
}

# Access the second value of the "LengthsCm" key
print(myFishDict["LengthsCm"][1])
```

```
# Access the first key's second value using values() and list conversion
print(list(myFishDict.values())[0][1])
```

```
# Dictionary constructor: dict() - create a dictionary using key=value pairs

myFishDict = dict(Species="Salmon", Habitat="River", LengthCm=75)

# Print the dictionary
print(myFishDict)
```

```
# Dictionary constructor: dict() - create a dictionary from a list of key-value pairs

myFishDict = dict([("Species", "Salmon"), ("Habitat", "River"), ("LengthCm", 75)])

# Print the dictionary
print(myFishDict)
```

```
# Length function: len() - returns the number of key-value pairs in a dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Get the number of items in the dictionary
print(len(myFishDict))
```

```
# Access the value of a key in a dictionary using key:value

myFishDict = dict(Species="Salmon", Habitat="River", LengthCm=75)

# Get the value associated with the "LengthCm" key
print(myFishDict["LengthCm"])
```

Nested dictionary

Description

Dictionaries can have a nested structure, where one or more values are themselves dictionaries. This allows dictionaries to store other dictionaries as values, enabling the representation of more complex, hierarchical structures. Nested dictionaries are particularly useful for organizing grouped data, such as records in a database or multi-level configuration settings. They provide both flexibility and clarity, making it easier to manage and access data across multiple layers.

Syntax

```
myDict = {
           Key1: {SubKey1: Value1, SubKey2: Value2},
           Key2: {SubKey1: Value3},
           ...
}
```

Examples

```
# Nested fish dictionary example

myFishDict =  { "Freshwater": {1: "Salmon", 2: "Trout", 3: { "a": ("Carp", "Catfish") } }, "Saltwater": { "a": ("T

# Accessing nested dictionary values
print(myFishDict["Saltwater"]["a"][0][0:3])
print(myFishDict["Saltwater"]["b"][:])

# Additional commented examples of accessing nested values:
# print(myFishDict)
# print(myFishDict["Freshwater"])
# print(myFishDict["Freshwater"][1])
# print(myFishDict["Freshwater"][3]["a"][1])
```

```
# print(myFishDict["Freshwater"][3]["a"][1][1:3])
# print(list(myFishDict.values()))
# print(list(myFishDict.values())[0][3]["a"][0])
# print(list(myFishDict.values())[1]["a"][1])
# print(list(myFishDict.values())[1]["b"])
```

Dictionary methods

Description

Dictionaries have several **built-in methods** that allow efficient interaction and manipulation of their data. The *get(key)* method retrieves the value for a given key, returning None if the key does not exist. The *keys()* and *values()* methods return views of all keys and values, respectively, while *items()* returns a view of **key-value pairs** as **tuples**. The *update()* method adds new key-value pairs or updates existing ones, and *pop(key)* removes a key along with its value, returning the removed value. Other useful methods include *clear()*, which removes all items, and *copy()*, which creates a shallow copy of the dictionary. Together, these methods provide flexible and powerful ways to manage and work with dictionaries efficiently.

Syntax

```
myDictionary.method()
```

Examples

```
# Get method: get() - access a dictionary value using its key

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Retrieve the value associated with the "LengthCm" key
print(myFishDict.get("LengthCm"))
```

```
# Keys method: keys() - get a view of all keys in the dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Retrieve all keys
print(myFishDict.keys())
```

```
# Values method: values() - get a view of all values in the dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Retrieve all values
print(myFishDict.values())
```

```
# Items method: items() - get a view of all key:value pairs in the dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Retrieve all key-value pairs
print(myFishDict.items())
```

```
# Update method: update() - update existing keys or add new key:value pairs

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Update the LengthCm key
myFishDict.update({"LengthCm": 80})

# Print the updated dictionary
print(myFishDict)
```

```
# Pop method: pop() - remove a key:value pair from the dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Remove the "Habitat" key and its value
myFishDict.pop("Habitat")

# Print the updated dictionary
print(myFishDict)
```

Dictionary operations

Description

Dictionary operations allow you to **add, update, access, and remove key-value pairs** efficiently. You can access a value by specifying its key, and update it by assigning a new value to an existing key. The pop(key) method removes a key-value pair and returns its value. Membership testing using in checks whether a key exists in the dictionary. To delete an entire dictionary, the **del statement can be used, which removes the dictionary from memory**, making it inaccessible. These operations provide a flexible and powerful way to work with structured data.

Syntax

- Create a dictionary: myDict = {"Key1": Value1, "Key2": Value2, "Key3": Value3}
- Access a value by key: value = myDict["Key1"]
- Update a value: myDict["Key2"] = NewValue
- Add a new key:value pair: myDict["Key4"] = Value4
- Remove a key:value pair using pop(): removedValue = myDict.pop("Key3")
- Membership testing: if "Key1" in myDict: print("Key exists")
- Delete the entire dictionary: del myDict

Examples

```
# Adding or updating a key-value pair in a fish dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Update the value of an existing key
myFishDict["LengthCm"] = 80

# Print the updated dictionary
print(myFishDict)
```

```
# Accessing a value by key in a fish dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Access the value associated with the "LengthCm" key
print(myFishDict["LengthCm"])
```

```
# Removing an item from a fish dictionary using pop()

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Remove the "Habitat" key and its value
myFishDict.pop("Habitat")

# Print the updated dictionary
print(myFishDict)
```

```
# Delete an item from a fish dictionary using del
```

```
myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Delete the "LengthCm" key and its value
del myFishDict["LengthCm"]

# Print the updated dictionary
print(myFishDict)
```

```
# Delete the entire fish dictionary

myFishDict = {"Species": "Salmon", "Habitat": "River", "LengthCm": 75}

# Delete the dictionary completely
# Trying to access it now will raise an error
# print(myFishDict)  # NameError: name 'myFishDict' is not defined

del myFishDict
```

Note

A Python dictionary can be easily loaded from a JSON file, a common and widely used format for representing structured data. JSON (JavaScript Object Notation) is a lightweight, human-readable file format primarily used for data exchange between a server and a web application. Its simplicity and ease of use make JSON ideal for storing and transmitting data across various programming languages. In Python, the json module provides functionality to parse JSON data and convert it into a Python dictionary. For example, a file named student.json can contain student-related information in JSON format, which can then be loaded into a Python dictionary for further processing.

The official JSON file specification can be found in RFC 8259 (https://datatracker.ietf.org/doc/html/rfc8259). This document provides the formal rules and guidelines for how JSON data should be structured. For Python-specific usage, the json module documentation is available at: https://docs.python.org/3/library/json.html.

Examples

```
# fish.json

{
    "Species": "Salmon",
    "Habitat": "River",
    "LengthCm": 75
}
```

```
# To load a JSON file into Python

import json

# Open and load the fish.json file
with open('fish.json', 'r') as jsonFile:
    fishDict = json.load(jsonFile)

# Print the loaded dictionary
print(fishDict)
```

3.4 Set

Description

A **set** is an **unordered collection** of **unique elements**, meaning it cannot contain duplicate values. Unlike lists or tuples, sets do not preserve the order of elements, making them ideal for **membership testing** and **eliminating duplicates**. Sets support a variety of operations, such as **union**, **intersection**, and **difference**, allowing for efficient mathematical set operations. Sets are **mutable**, so you can add or remove elements after creation, but their elements must be **immutable** (e.g., numbers, strings, or tuples). Sets are commonly used when working with **distinct items** and performing operations like filtering, searching, or removing duplicates from a collection of data.

Set declarations

Description

A **set** can be declared in two ways: using **curly braces** {} or the *set()* constructor. To create a set with curly braces, list the elements separated by commas inside the braces. Alternatively, the *set()* **constructor** can be used with a **list** or **tuple** inside the parentheses, which is especially useful for converting other data types into a set. A key characteristic of sets is that they **automatically eliminate duplicate values**, so the result always contains only **unique elements**. **Sets do not maintain order**, so elements may appear in an **arbitrary order** each time you access the set.

Syntax

```
mySet = {element 1, element 2, ..., element n}

mySet = {element 1}

mySet = set()
```

Examples

```
# Creating an empty set correctly
# Note: {} creates an empty dictionary, not a set

myEmptySet = set()  # use set() constructor for an empty set

print(myEmptySet)
print(type(myEmptySet))
```

```
# Creating an empty set using the set() constructor

myEmptySet = set()

print("My empty set:", myEmptySet)
print(type(myEmptySet))
```

```
# Creating a set with multiple string values

myFishSet = {"Carp", "Bass", "Salmon", "Red Snapper", "Pike", "Perch"}

# Print the set
print(myFishSet)
```

```
# Creating a set with multiple string values using the set() constructor

myFishSet = set(("Carp", "Bass", "Salmon", "Red Snapper"))  # Use tuple inside set()

# Print the set
print(myFishSet)
```

```
# Creating a set with duplicate values

myFishSet = {"Carp", "Bass", "Salmon", "Red Snapper", "Bass", "Bass", "Bass"}

# Print the set
print(myFishSet)
```

```
# Creating a set with mixed types: strings, booleans, and integers

myFirstSet = {"Carp", "Bass", "Red Snapper", "Bass", True, 1, 2}

# Print the set
print(myFirstSet)
```

```
# Sets with different data types

# Set of strings
myFishSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Set of integers (duplicates automatically removed)
myNumberSet = {1, 2, 3, 4, 5, 4, 6, 3, 21}

# Set of booleans (duplicates automatically removed)
myBooleanSet = {True, False, False, True, False}

# Mixed set: strings, integers, booleans
myMixedSet = {"Carp", True, 23, 40, "Bass", "Salmon"}

# Set of complex numbers
myComplexSet = {2+3j, 1+7j, 3+4j}

# Print all sets
print(myFishSet)
print(myNumberSet)
print(myBooleanSet)
print(myMixedSet)
print(myComplexSet)
```

Set methods

Description

Python sets provide a variety of built-in methods to efficiently manage unique elements. You can add elements using *add()* for a single item or *update()* to add multiple items from another iterable (like a list or another set). Elements can be removed with *remove()* (raises an error if the element doesn't exist) or discard() (does not raise an error). The *pop()* method removes and returns an arbitrary element. Membership testing is done using the in keyword. Additional methods include *clear()* to remove all elements, and *copy()* to create a shallow copy of a set.

Sets also support mathematical operations:

union() – combines two sets and returns a new set with all unique elements.

intersection() – returns a set containing only elements common to both sets.

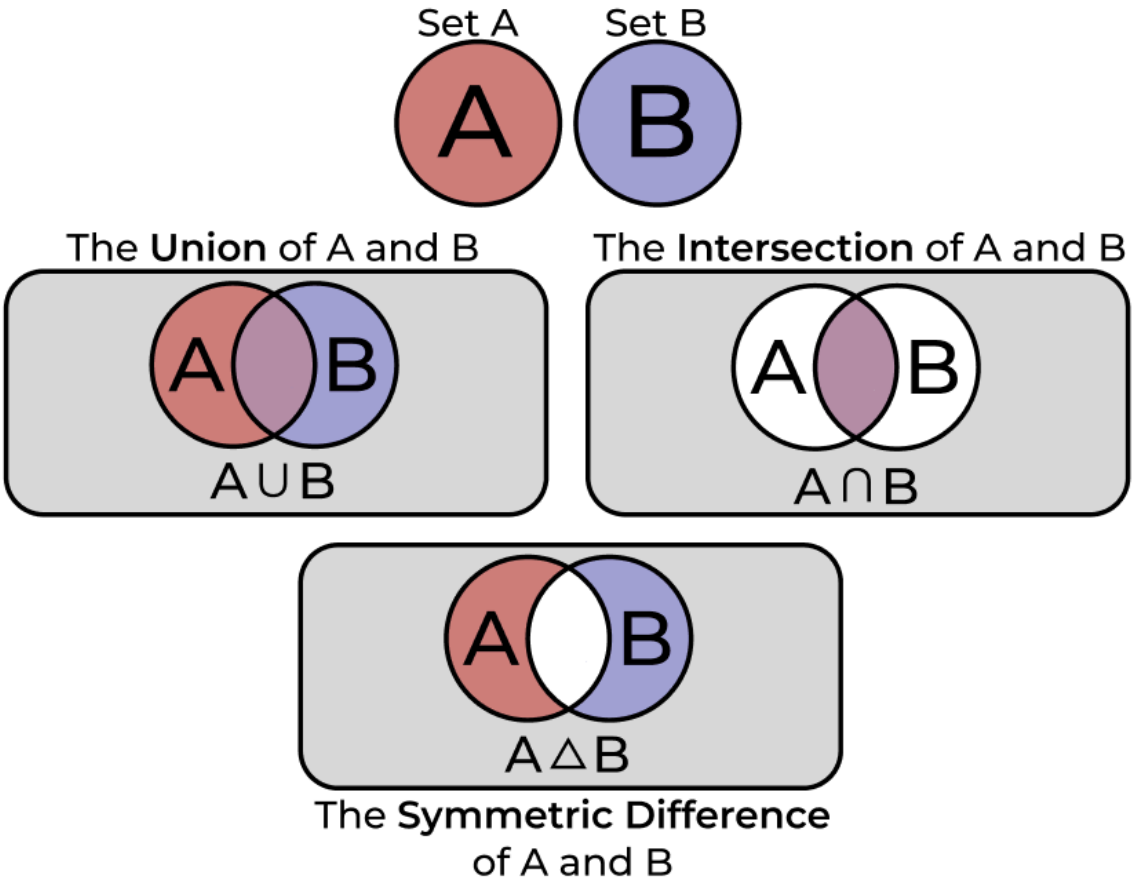
difference() – returns elements that are in the first set but not in the second.

symmetric_difference() – returns elements present in either set but not in both.

These operations make sets ideal for comparing, merging, and filtering collections of unique items efficiently.

Note

Python sets are conceptually similar to mathematical sets in many ways. Just like mathematical sets, Python sets are collections of unique elements with no particular order. You can perform the same fundamental operations on Python sets as you would with mathematical sets.



Syntax

```
mySet.method()
```

Examples

```
# Add method: add() - adds a single element to the set

myFishSet = {"Carp", "Bass", "Red Snapper"}

# Add a new element
myFishSet.add("Salmon")

# Print the updated set
print(myFishSet)
```

```
# Update method: update() - adds multiple elements from another set or iterable

myFirstSet  = {"Carp", "Bass"}
mySecondSet = {"Red Snapper", "Salmon"}

# Add all elements from mySecondSet to myFirstSet
myFirstSet.update(mySecondSet)

# Print the updated set
print(myFirstSet)
```

```
# Update method: update() - adds multiple elements from a list or any iterable

myFirstSet   = {"Carp", "Bass"}
mySecondList = ["Red Snapper", "Salmon"]

# Add all elements from mySecondList to myFirstSet
myFirstSet.update(mySecondList)
```

```
# Print the updated set
print(myFirstSet)
```

```
# Remove method: remove() - removes a specified element from the set
# Raises a KeyError if the element does not exist

myFirstSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Remove the element "Carp" from the set
myFirstSet.remove("Carp")

# Print the updated set
print(myFirstSet)
```

```
# Discard method: discard() - removes a specified element from the set
# Does not raise an error if the element is not found

myFirstSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Remove the element "Red Snapper" from the set
myFirstSet.discard("Red Snapper")

# Print the updated set
print(myFirstSet)
```

```
# Pop method: pop() - removes and returns an arbitrary element from the set
# Since sets are unordered, you do not know which element will be removed

myFirstSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Remove an arbitrary element from the set
myFirstSet.pop()

# Print the updated set
print(myFirstSet)
```

```
# Clear method: clear() - removes all elements from the set

myFirstSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Empty the set
myFirstSet.clear()

# Print the empty set
print(myFirstSet)
```

```
# Intersection method: intersection() - returns a set containing elements common to both sets

myFirstSet  = {"Carp", "Red Snapper"}
mySecondSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Find elements common to both sets
myThirdSet = myFirstSet.intersection(mySecondSet)

# Print the resulting set
print(myThirdSet)
```

```
# Intersection update method: intersection_update() - updates the set to keep only elements common to both sets

myFirstSet  = {"Carp", "Bass", "Red Snapper", "Salmon"}
mySecondSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# Update myFirstSet to contain only elements also in mySecondSet
myFirstSet.intersection_update(mySecondSet)

# Print the updated set
print(myFirstSet)
```

```
# Symmetric difference update method: symmetric_difference_update()
# Updates the set to keep elements that are in either set, but not in both

myFirstSet  = {"Carp", "Bass", "Red Snapper", "Salmon"}
mySecondSet = {"Red Snapper", "Salmon"}

# Update myFirstSet to keep only elements that are not in both sets
myFirstSet.symmetric_difference_update(mySecondSet)

# Print the updated set
print(myFirstSet)
```

```
# Symmetric difference method: symmetric_difference()
# Returns a new set with elements that are in either set but not in both

myFirstSet  = {"Bass", "Red Snapper", "Salmon"}
mySecondSet = {"Carp", "Red Snapper", "Salmon"}

# Create a new set containing only elements unique to each set
myThirdSet  = myFirstSet.symmetric_difference(mySecondSet)

print(myThirdSet)
```

```
# Union method: union() – joins two sets while excluding duplicates

myFirstSet  = {"Red Snapper", "Salmon"}
mySecondSet = {"Carp", "Bass"}

# Create a new set containing all elements from both sets without duplicates
myThirdSet = myFirstSet.union(mySecondSet)

print(myThirdSet)
```

Set operations

Description

Python sets support a few basic built-in operations similar to other collections like lists or strings. You can use the *len()* function to get the number of elements in a set, and the **in** membership operator to check if an element exists in the set. The **del** statement can be used to delete the entire set from memory. Since sets are unordered, Python does not support **indexing** ([index]), **slicing** ([start:end:step]), **concatenation** (+), or **repetition** (*) operations on sets. Attempting to perform these operations will result in a **TypeError**. These restrictions emphasize the mathematical nature of sets, which focus on uniqueness and unordered elements rather than sequence and repetition.

Syntax

- Create a set: MySet = {Element1, Element2, Element3}
- Length of the set: SetLength = len(MySet)
- Membership testing: IsPresent = Element1 in MySet; IsAbsent = Element4 not in MySet
- Add an element: MySet.add(NewElement)
- Update set with multiple elements: MySet.update([Element4, Element5])
- Remove an element: MySet.remove(Element2) - raising an error
- Remove an element safely: MySet.discard(Element3)
- Remove and return an arbitrary element: RemovedElement = MySet.pop()
- Clear all elements: MySet.clear()
- Set operations:

SetUnion = MySet1.union(MySet2)

SetIntersection = MySet1.intersection(MySet2)

SetDifference = MySet1.difference(MySet2)

SetSymDiff = MySet1.symmetric_difference(MySet2)

- Update set in-place with intersection: MySet1.intersection_update(MySet2)
- Update set in-place with symmetric difference: MySet1 symmetric_difference_update(MySet2)
- Delete the entire set: del MySet

Examples

```
# Obtaining the length of the set using len()

mySecondSet = {"Carp", "Bass", "Red Snapper", "Salmon", "Bass"}  # duplicates are ignored

print(len(mySecondSet))
```

```
# Checking membership in a set using the `in` operator
mySecondSet = {"Carp", "Bass", "Red Snapper", "Salmon", "Bass"}  # duplicates are ignored

# Check if "Carp" is in the set
print("Carp" in mySecondSet)
```

```
# Delete the entire set using the del statement

myFishSet = {"Carp", "Bass", "Red Snapper", "Salmon"}

# myFishSet no longer exists after this
del myFishSet
```

3.5 Summary of data collections

Description

```
String          List        Tuple      Dictionary / Set

immutable     mutable      immutable        mutable
order/index order/index  order/indexed     unordered
duplicates    duplicates   duplicates     noduplicates
  "   "        [   ]        (  )       {   } / {*}, set()
  str()        list()      tuple()       dict() / set()

{*} - A set can not be declared with empty braces
```

Note

Python is an **object-oriented** programming language, meaning that objects are central to its design. Everything in Python is treated as an object, which contains both **data (attributes)** and **functionality (methods)**. Objects are created using a **constructor**, a **special method within a class** that initializes and builds instances of that class. A **class** acts as a blueprint defining the structure and behavior of its objects. Once an object (or instance) is created from a class, it can hold specific data and functionality, making classes a powerful and reusable programming construct.

```
class Fish:
    # Constructor
    def __init__(self, species, length, color):
        self.species = species    # attribute for species
        self.length  = length     # attribute for length in cm
        self.color   = color      # attribute for color

# Creating an object (instance) of the Fish class
myFish = Fish("Carp", 35, "Golden")
```

```
# Accessing attributes of the object
print("Fish species:", myFish.species)
print("Fish length (cm):", myFish.length)
print("Fish color:", myFish.color)
```

Section 4: Control flow

4.1 if ... statements

4.2 For loops

4.3 While loops

4.1 if ... statements

Description

The **if** statement is used to execute code based on specific conditions, enabling programs to make decisions. It is one of the most fundamental forms of decision-making in programming. Conditional statements come in three main forms: a simple **if** to run code when a condition is true, **if-else** to provide an alternative block when the condition is false, and **if-elif-else** to check multiple conditions sequentially, executing the first block whose condition evaluates to true.

Syntax

```
if condition: block of code

if condition: block of code else: block of code

if condition: block of code elif condition: block of code else: block of code
```

if ... statement

Description

The syntax of an if statement begins with the if keyword, followed by a condition, and ends with a colon. The indented block of code underneath runs only if the condition evaluates to True. A simple if statement can also be written in a single line, which functions the same as the multi-line version but is best suited for short, straightforward actions.

Syntax

```
    if condition: block of code
```

Examples

```
# If statement in two lines

if True:
  print ("Hello, World!")
```

```
# If statement in one line

if True: print("Hello, World!")
```

```
# If statement with the value of 1

if 1: print("Hello, World!")
```

```
# If statement with the value of 0

if not 0: print("Hello, World!")
```

```
# To reserve the if statement while you are writing a program

if True: pass
```

if ... statement with comparison operators

Description

if statements are used to check **conditions** in a program. The basic form evaluates a condition using **comparison operators** like **equal to (==)**, **not equal to (!=)**, **greater than (>)**, **less than (<)**, **greater than or equal to** **(>=)**, and **less than or equal to (<=)**. If the **condition** is true, the indented block of code under the if statement is **executed**. For example, a program could check the type of fish and print a message only if it matches a specific value. For more details, see Section 2.

Syntax

```
    if comparison statement: block of code
```

Examples

```
# A simple example of an if statement with a comparison operator
# Comparison operators: ==, !=, >, <, >=, <=

myVariable = 1

if myVariable == 1: print("My integer variable is:", myVariable)
```

```
# A simple example of an if statement with a comparison operator
# Comparison operators: ==, !=, >, <, >=, <=

myVariable = -1

if myVariable != 1: print("My integer variable is:", myVariable)
```

```
# Integer and float comparison
myInt = 1
myFloat = 1.0

if myInt == myFloat: print("Integer and float are equal")
```

```
# A simple example of an if statement with a comparison operator
# Comparison operators: ==, !=, >, <, >=, <=

# Check if a float is greater than a threshold
myFloat = 1.0
```

```
if myFloat > 0.99: print("True")
```

```
# Comparing floats safely using a small tolerance

myFloat1 = 1.0
myfloat2 = 0.99999
tolerance = 1e-5

if abs(myFloat1 - myFloat2) < tolerance: print("Almost equal")
```

Note

You can directly compare integers and floats in Python since both are numeric types. For instance, 1 == 1.0 evaluates to True. However, when comparing floating-point numbers that are very close in value, rounding errors can cause unexpected results. For example, 1.0 is indeed greater than 0.99999, but direct equality checks might fail for values that are nearly identical. To safely compare floats for approximate equality, it's recommended to use a small tolerance, such as if abs(1.0 - 0.99999) < 1e-5, which ensures the numbers are considered "almost equal."

if ... statement with arithmetic, comparison, and assignment operators

Description

if statements can include expressions that combine arithmetic and comparison operators to form conditions. Arithmetic operators (see Section 2) can be used to perform calculations within the condition, and comparison operators are used to evaluate the results of those calculations. Regular assignment operators (=) **cannot be used directly inside the condition**; the assignment must be done beforehand.

The **walrus operator (:=)** is a **special assignment** expression that allows you to assign a value to a variable as part of an expression, including inside an if statement. This operator combines assignment and evaluation in a single step. It is particularly useful when you want to both **use** and **test** the result of an expression in the same line.

Syntax

```
    if arithmetic comparison: block of code
```

Examples

```
# if ... statements with arithmetic ( % ) and comparison ( == ) operators

myFirstVariable  = 20

if ( myFirstVariable % 2 == 0 ): print("myFirstVariable is an even number" )
```

```
# if ... statements with arithmetic ( // ) and comparison ( == ) operators

mySecondVariable = 19

if (mySecondVariable // 2 == 9): print("Floor division rounds the result down to the nearest whole number")
```

```
# if x = 5: print('x is 5') Illegal statement since assignment operator (=) is used

if (x:=5) > 4: print(f"x is {x} and greater than 4")
```

```
# if statement combining arithmetic (**), comparison (>=), and walrus (:=) operators

baseValue = 2

# Compute baseValue to the power of 8 and check if it is greater than or equal to 256
if (result := baseValue ** 8) >= 256: print("Exponentiation test:", result)
```

if ... statement with arithmetic, comparison, and logical operators

Description

if statements in Python can combine **arithmetic**, **comparison**, and **logical operators** to create complex conditions. Arithmetic operators perform calculations within the condition, comparison operators evaluate relationships between values, and logical operators such as and, or, and not allow multiple conditions to be combined. This combination enables powerful decision-making based on several factors in a single if statement. For example, **if (x + y > z) and (z % y == 0):** uses **arithmetic** and **comparison** together with a **logical** operator to determine if both conditions are true before executing the code block.

Syntax

```
if arithmetic comparison logical ... : block of code
```

Examples

```
# if ... statement with arithmetic (%), comparison (==), and logical (and) operators

myFirstNumber  = 1
mySecondNumber = 2

if myFirstNumber % 2 == 0 and mySecondNumber % 2 == 0:
    print("Both numbers are even")
```

```
# if ... statement with arithmetic (%), comparison (==), and logical (or) operators

myFirstNumber  = 1
mySecondNumber = 2

if myFirstNumber % 2 == 0 or mySecondNumber % 2 == 0:
    print("One condition is sufficient to be True")
```

```
# if ... statement with arithmetic (%), comparison (==), and logical (not) operators

myFirstNumber = 20

if not myFirstNumber % 2 == 1:
    print("The condition is reversed")
```

Nested if ... statement

Description

Nested **if** statements in Python allow you to check multiple conditions in a hierarchical manner, where an inner condition is evaluated only **if** the outer condition is true. They are useful when decisions depend on the outcome of previous conditions, helping to create clear and structured logic. However, they should be used with care to avoid overly deep nesting, which can make code harder to read and maintain.

Syntax

```
if condition:
    if condition:
        block of code
```

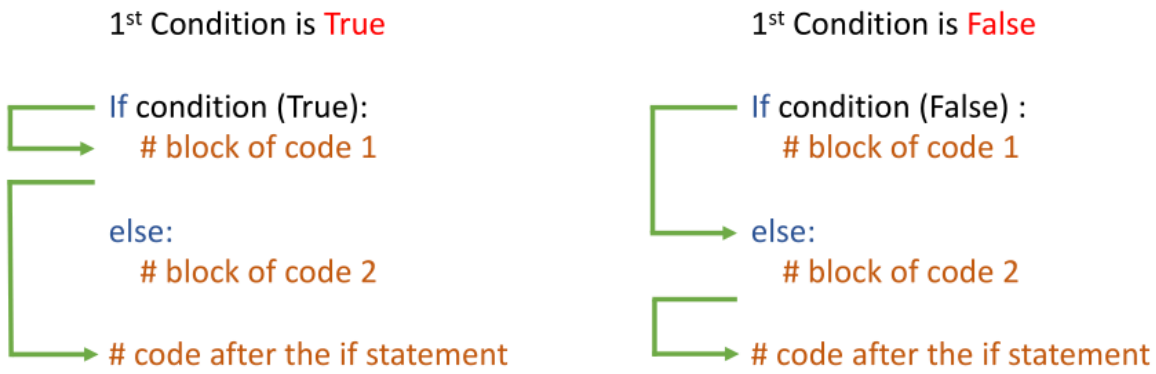
Examples

```
# Nested if statements example

myFirstNumber  = 1
mySecondNumber = 2

if myFirstNumber < mySecondNumber:
    if mySecondNumber == 2:
        print("Both conditions are True")
```

```
# Nested if statements

myFirstNumber  = 1
mySecondNumber = 2
myThirdNumber  = 3

if myFirstNumber == 1:
    print("My first number is:", myFirstNumber)
    if mySecondNumber == 2:
        print("My second number is:", mySecondNumber)
        if myThirdNumber == 3:
            print("My third number is:", myThirdNumber)
```

if ... else statement

if ... else statements are used to execute one block of code when a condition is **True** and a different block of code when the condition is **False**. This allows a program to handle two possible outcomes effectively. The if ... else statement is a fundamental part of decision-making in Python programming.



Syntax

```
if condition:
    block of code (if condition is true)
else:
    block of code (if condition is false)
```

if ... else statement with comparison operators

Description

if...else statements can be combined with **comparison operators** to make decisions based on how values relate to each other. Depending on the result of the comparison, the program executes **one** of **two code blocks**. This structure is essential for controlling program flow and enabling the program to respond appropriately to different conditions.

Syntax

```
    See examples below
```

Examples

```
# Equal to comparison operator: ==

myFirstNumber  = 0.0
mySecondNumber = 1.0

if myFirstNumber == mySecondNumber:
    print("First condition is True")
else:
    print("First condition is False")
```

```
# Not equal to comparison operator: !=

myFirstNumber  = 1.0
mySecondNumber = 1.0

if myFirstNumber != mySecondNumber:
    print("First condition is True")
else:
    print("First condition is False")
```

```
# Greater than comparison operator: >

myFirstNumber  = 1.0
mySecondNumber = 1.0

if myFirstNumber > mySecondNumber:
    print("First condition is True")
else:
    print("First condition is False")
```

```
# Less than comparison operator: <

myFirstNumber  = 1.0
mySecondNumber = 1.0

if myFirstNumber < mySecondNumber:
    print("First condition is True")
else:
    print("First condition is False")
```

```
# Greater than or equal to comparison operator: >=

myFirstNumber  = 1.0
mySecondNumber = 1.0

if myFirstNumber >= mySecondNumber:
    print("First condition is True")
else:
    print("First condition is False")
```

```
# Less than or equal to comparison operator: <=

myFirstNumber  = 1.0
mySecondNumber = 1.0

if myFirstNumber <= mySecondNumber:
    print("First condition is True")
else:
    print("First condition is False")
```

Syntax

```
  block of code (if condition is true) if condition else block of code (if condition is false)
```

```
# This is called a ternary conditional operator or short-hand if...else in Python.

myFirstVariable  = 1.0
mySecondVariable = 2.0

print("True") if myFirstVariable == mySecondVariable else print("False")
```

if ... else statement with arithmetic and comparison operators

Description

if...else statements can use both arithmetic and comparison operators to make decisions based on calculated values. Arithmetic operators perform mathematical operations, while comparison operators evaluate the results. By combining these operators within if...else statements, you can create conditional logic that reacts to the outcome of mathematical calculations and comparisons.

Syntax

```
if (arithmetic expression comparison operator value):
    block of code executed (if condition is True)
else:
    block of code executed (if condition is False)
```

Examples

```
# if ... else statement with arithmetic (+) and comparison (==) operators

myNumber = 1

if myNumber + myNumber == 2:
    print("The value is 2")
else:
    print("The value is not 2")
```

```
# if ... else statement with arithmetic (%) and comparison (==) operators

myNumber = 1

if myNumber % 2 == 0:
    print("The number is even")
else:
    print("The number is odd")
```

```
# if ... else statement with arithmetic (//) and comparison (==) operators

myNumber = 1

if myNumber // 1 == 1:
    print("Floor division result is 1")
else:
    print("Floor division result is another number:", myNumber // 1)
```

```
# Short-hand if...else with walrus operator

myNumber  = 1
myLimit   = 2

print("True") if (result := myNumber + 1) <= myLimit else print("False")
```

if ... else statement with arithmetic, comparison, and logical operators

Description

if...else statements can combine arithmetic, comparison, and logical operators to create complex decision-making conditions. By using these operators together, you can build conditional logic that reacts to mathematical calculations, value comparisons, and multiple logical conditions within a single statement.

Syntax

```
if (arithmetic expression comparison operator value) logical operator (arithmetic expression comparison operator
    block of code executed (if condition is True)
else:
    block of code executed (if condition is False)
```

Examples

```
# if ... else statement with arithmetic (%), comparison (==), and logical (and) operators

myNumber1 = 1
myNumber2 = 2

if myNumber1 % 2 == 0 and myNumber2 % 2 == 0:
    print("Both numbers are even")
else:
    print("At least one number is odd")
```

```
# if ... else statement with arithmetic (+), comparison (>), and logical (or) operators

myNumber1 = 1
myNumber2 = 2

if myNumber1 + myNumber2 > 2 or myNumber2 > 2:
    print("One of the conditions is True")
else:
    print("None of the conditions is True")
```

```
# if ... else statement with arithmetic (*), comparison (==), and logical (not) operators

myNumber1 = 1
myNumber2 = 2

if not myNumber1 * 2 == myNumber2:
    print("The condition is True")
else:
    print("The condition is False")
```

```
# Short-hand if...else (ternary conditional operator) example

myNumber1 = 1
myNumber2 = 2

print("True") if myNumber1 <= myNumber2 and myNumber1 > 0 else print("False")
```

Nested if ... else statement

Nested **if...else** statements allow you to place one **if...else** block inside another, enabling the program to make more specific decisions based on multiple levels of conditions. This structure is useful when a decision depends on the outcome of a previous condition. It allows you to create more complex conditional logic and evaluate multiple conditions in a structured way. Below are two examples of nested **if...else** statements.

Syntax

```
if condition1:
   block of code executed (if condition1 is True)

   if condition2:
      block of code executed (if condition2 is True)
   else:
      block of code executed (if condition2 is False)
else:
   block of code executed (if condition1 is False)
```

Examples

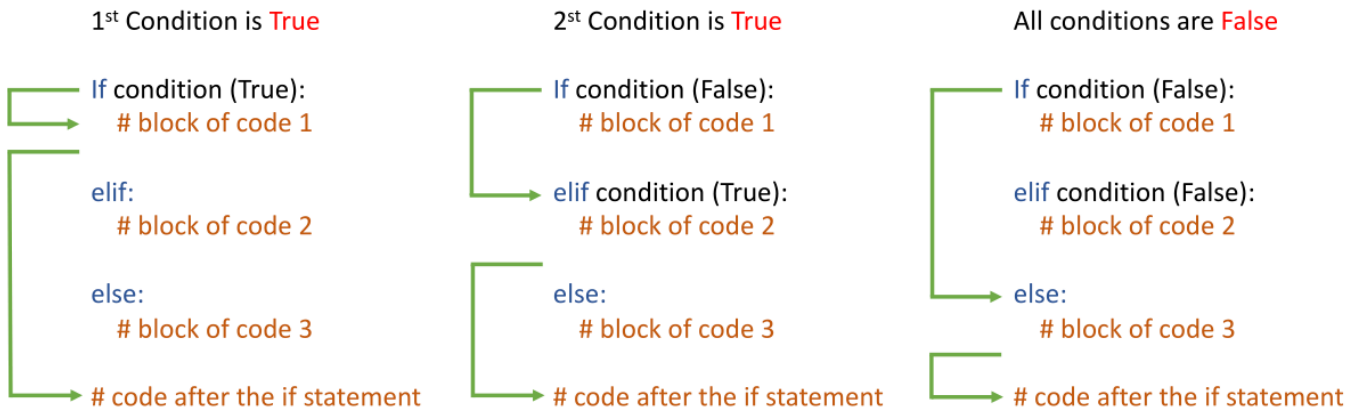
```
# Nested if...else statement example

myNumber1 = 1
myNumber2 = 2

if myNumber1 == 1:
    print("My first number is:", myNumber1)

    if myNumber2 == 1:
        print("My second number is 1")
    else:
        print("My second number is not 1, it is:", myNumber2)
```

```
# Nested if...else statement example

myChar1 = "A"
myChar2 = "B"

if myChar1 == "A":
    print("My first character is:", myChar1)

    if myChar2 == "B":
        print("My second character is:", myChar2)
    else:
        print("My second character is not B, it is:", myChar2)
else:
    print("My first character is not A, it is:", myChar1)
```

if ... elif ... else statement

Description

The **if...elif...else** statement is used to handle multiple conditions in a clear and structured way. It begins with an **if condition**, followed by one or more **elif** (short for "else if") conditions, and can end with an optional **else block**. The program evaluates each condition in order and executes the block of code for the first condition that is True, skipping the remaining conditions. If none of the conditions are True, the else block runs. This structure is ideal for selecting one outcome from several possibilities without resorting to multiple nested if statements.



Syntax

```
if condition1:
    block of code
elif condition2:
    block of code
else:
    block of code
```

if ... elif ... else statement with comparison operators

Description

The **if...elif...else** statement can be combined with comparison operators to make decisions based on multiple conditions. Comparison operators evaluate the relationship between values, allowing the program to execute different code blocks depending on the results. Conditions are evaluated in order, and only the first True condition executes. If none of the conditions are True, the else block is executed, providing a default action.

Syntax

```
See examples below
```

Examples

```
# if...elif...else statement with comparison operators

myNumber = 2

if myNumber == 1:
    print("The number is 1")
elif myNumber == 2:
    print("The number is 2")
else:
    print("The number is something else")

print("Out of if...elif...else statement")
```

```
# if...elif...else statement with comparison (==) and (<) operators
# Second condition is True

myChar1 = "a"
myChar2 = "b"

if myChar1 == "A":
```

```
    print("First condition is True")
elif myChar1 < myChar2:
    print("Second condition is True")
else:
    print("All conditions are False")

print("Out of if...elif...else statement")
```

```
# if...elif...else statement with comparison (==) and (>) operators

myNumber1 = 0
myNumber2 = 2

if myNumber1 == 0:
    print("My first number is zero")
elif myNumber1 > 1:
    print("My first number is greater than one")
elif myNumber1 == myNumber2:
    print("My first number equals my second number")
else:
    print("My first number is less than zero")

print("Out of if...elif...else statement")
```

if ... elif ... else statement with comparison and logical operators

Description

The **if...elif...else** statement can be enhanced with **comparison** and **logical** operators to create more complex decision-making conditions. Comparison operators evaluate relationships between values, while logical operators allow multiple conditions to be combined. This structure enables the program to handle several criteria and execute the appropriate code block based on more refined conditions.

Syntax

```
    See examples below
```

Examples

```
# if...elif...else statement with comparison (==, >, >=) and logical (and, or, not) operators

myNumber1 = 2
myNumber2 = 2

if myNumber1 == myNumber2 and myNumber1 < 0:
    print("Both conditions must be True")
elif myNumber1 > myNumber2 or myNumber1 > 2:
    print("At least one condition is True")
elif not myNumber1 >= myNumber2:
    print("The not condition is True")
else:
    print("All conditions are False")
```

Nested if ... elif ... else statement

Description

A nested **if...elif...else** statement is a control structure in which an **if**, **elif**, or **else block** is placed inside another. This structure allows the program to evaluate multiple levels of conditions. Nested **if...elif...else** statements are useful for handling more detailed or hierarchical decisions, but they should be used carefully to avoid creating overly complex structures that are hard to read and maintain.

Syntax

```
    See examples below
```

Examples

```
# Nested if...elif...else statements with comparison operators

firstVal = 1
secondVal = 2
thirdVal = 3

if firstVal < secondVal:
    if firstVal > thirdVal:
        print("firstVal is less than secondVal but greater than thirdVal")
    elif firstVal == thirdVal:
        print("firstVal is less than secondVal but equal to thirdVal")
    else:  # firstVal < thirdVal
        print("firstVal is less than both secondVal and thirdVal")

elif firstVal == secondVal:
    print("firstVal is equal to secondVal")

else:  # firstVal > secondVal
    print("firstVal is greater than secondVal")
```

4.2 For loops

Description

A **for loop** is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code repeatedly for each item in that sequence. It is one of the most commonly used loops in Python due to its simplicity and readability. The loop begins with the for keyword, followed by a variable representing the current item, and the in keyword to specify the sequence. The block of code to be executed is indented beneath the loop. A for loop is particularly useful when the number of iterations is known in advance or when you want to process every item in a collection without manually indexing each element.

Syntax

```
    for item in sequence:
        block of code
```

For loop

Description

A **for loop** is an iterative construct that allows you to process each element in a data sequence. It uses two keywords: for and in. During each iteration, the loop accesses the current element of the sequence. The loop continues until it has processed the last element of the sequence.

Syntax

```
    See examples below
```

Examples

```
# A for loop example with a dummy pass statement

for i in range(10): pass
```

```
# A for loop example with the range() function

for i in range(10): print(i, end=" ")
```

```
# A for loop example using a list of fish

myFishList = ["Bass", "Red Snapper", "Karp", "Pike", "Perch"]

for name in myFishList: print(name, end="; ")
```

For loop with no elements

Description

In a **for loop**, an underscore (_) is commonly used as a placeholder for values you want to ignore. This is a convention when you need to iterate over a sequence but do not need to use the individual elements. Using _ makes it clear that the loop variable is intentionally unused.

Syntax

```
for _ in sequence:
   block of code
```

Example

```
# Using underscore as a throwaway variable in a for loop

myNumberList = range(3)

for _ in myNumberList: print("Hello World!")
```

For loops with data sequences

Description

For loops are commonly used to iterate over data sequences such as **strings**, **lists**, **tuples**, **dictionaries**, and **sets**. These sequences contain **multiple elements**, and a **for loop allows you to process each element one at a time**. For example, you can loop through a string character by character or through lists, tuples, and other structures in a similar way. Below are a few examples of for loops with different data sequences.

Syntax

```
for item in sequence:
    block of code
```

Examples

```
# A for loop with a string

for char in "Fishing Trip, July 5th, 2025": print(char, end=" ")
```

```
# A for loop with a number list sequence

myNumberList = [1, 2, 3, 4, 5]

for num in myNumberList: print(num, end=" ")
```

```
# A for loop with a number tuple sequence

myNumberTuple = (1, 2, 3, 4, 5)

for num in myNumberTuple: print(num, end=" ")
```

```
# A for loop with a dictionary's keys

myFishCatch = {
    "Bass": 5,
    "Trout": 3,
    "Salmon": 7,
    "Pike": 2,
    "Perch": 4
}

for fish in myFishCatch: print(fish, end=" ")
```

```
# A for loop with a dictionary's values (fishing-related)

myFishCatch = {
    "Bass": 5,
    "Trout": 3,
    "Salmon": 7,
    "Pike": 2,
    "Perch": 4
}

for fish in myFishCatch: print(myFishCatch[fish], end=" ")
```

```
# A for loop with a set sequence

myNumberSet = {1, 2, 3, 4, 5}

for num in myNumberSet: print(num, end=" ")
```

Dictionary methods with For loops

Description

You can use for loops with dictionary methods to efficiently iterate over **keys**, **values**, or **both**. Dictionaries store data in **key-value pairs**, and **looping through them allows you to process or analyze each pair**. The **keys()** method loops through just the **keys**, the **values()** method loops through the **values**, and the **items()** method allows you to access both keys and values simultaneously. These methods make it easy to navigate and manipulate dictionary data, which is especially useful when working with structured information such as settings, records, or grouped data.

Syntax

```
for item in myDictionary.method():
    block of code
```

Examples

```
# A for loop with a dictionary's keys method

myFishCatch = {
    "Bass": 5,
```

```
    "Trout": 3,
    "Salmon": 7,
    "Pike": 2,
    "Perch": 4
}

for fish in myFishCatch.keys(): print(fish, end=" ")
```

```
# A for loop with a dictionary's values method

myFishCatch = {
    "Bass": 5,
    "Trout": 3,
    "Salmon": 7,
    "Pike": 2,
    "Perch": 4
}

for quantity in myFishCatch.values():
    print(quantity, end=" ")
```

```
# A for loop with a dictionary's items method (fishing-related)

myFishCatch = {
    "Bass": 5,
    "Trout": 3,
    "Salmon": 7,
    "Pike": 2,
    "Perch": 4
}

for fish, quantity in myFishCatch.items():
    print(f"{fish}: {quantity}", end="; ")
```

For loop with else statement

Description

A **for loop** can be **paired** with an **else** statement, which executes after the loop finishes iterating over all items. **This is often used when searching for an item and performing an action if the item is not found.** The **for...else** structure provides a clean way to handle post-loop logic when no early exit occurs.

Syntax

```
for item in sequence:
    block of code
else
    block of code
```

Examples

```
# For loop with else statement

myNumberList = [1, 2, 3, 4, 5]

for num in myNumberList:
    print(num, end=" ")
else:
    print("\nThis sequence is over!")
```

For loop with break, continue, and pass statements

Description

For loops can be controlled more precisely using the **break**, **continue**, and **pass** statements. The **break statement immediately exits the loop** when a certain condition is met, stopping any further iterations. The **continue statement skips the rest of the current iteration **and moves to the next item in the sequence, allowing selective execution of loop code. The **pass statement does nothing and serves as a placeholder** when a statement is syntactically required but no action is needed. These control statements make loop behavior more flexible and responsive to specific conditions.

Syntax

```
Using break statement

for item in sequence:
    if condition:
        break
    block of code to execute if condition is False

Using continue statement

for item in sequence:
    if condition:
        continue
    block of code to execute if condition is False

Using pass statement

for item in sequence:
    if condition:
        pass
    block of code to execute regardless the condition status
```

Examples

```
# A for loop with a break statement

myFirstList = [1, 2, 3, 4, 5]

for i in myFirstList:
    if i == 3:
        break
    print("i =", i)
```

```
# A for loop with the % operator and a continue statement

myFirstList = [1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in myFirstList:
    if i % 2 == 0:
        continue
    print(i, end=" ")
```

```
# A for loop with the % operator and a pass statement

myFirstList = [1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in myFirstList:
    if i % 2 == 0:
        pass  # Placeholder, does nothing for even numbers
    print(i, end=" ")
```

Nested For loop

Description

A nested **for loop** is a **for loop placed inside another for loop**. It allows you to iterate through multi-dimensional lists or matrices. Nested for loops are **useful for tasks such as iterating through rows and columns of a 2D list**, comparing each element in one list to every element in another, or generating combinations of multiple variables. Here's an example of a nested for loop.

Syntax

```
for item in sequence1:
    for item in sequence2:
        block of code
```

```
# A nested for loop

myFirstList = [1, 2, 3]
mySecondList = [1, 2, 3]

for i in myFirstList:
    for j in mySecondList:
        print(i, j, end=" ")
```

Nested for loop with break statements

Description

A **for loop** with a **break statement** allows you to exit the loop prematurely when a specific condition is met. Normally, a for loop iterates over all elements in a sequence, but **break stops the loop as soon as the desired result is found** or the stopping condition is triggered. This is particularly useful in search operations, as continuing the loop becomes unnecessary. Using break can improve efficiency by avoiding extra iterations once the goal is achieved.

Syntax

```
for item in sequence1:
    for item in sequence2:
        break
```

Examples

```
# A nested for loop with an innermost break statement

myFirstList  = [1, 2, 3]
mySecondList = [1, 2, 3]

for i in myFirstList:
    for j in mySecondList:
        if j == 3:
            break  # Exit the inner loop when j equals 3
        print(i, j, end=" ")
```

```
# A nested for loop with an outer break statement

myFirstList    = [ 1, 2, 3 ]
mySecondList   = [ 1, 2, 3 ]

for i in myFirstList:
  if i == 3:
      break
  for j in mySecondList:
    print (str(i),str(j)," ")
```

Nested for loop with continue statements

Description

A nested **for loop** with a continue statement allows you to skip specific iterations within the **inner** or **outer loop** based on a condition. The continue statement immediately moves to the next iteration of the current loop level without executing the remaining code in that iteration. In nested for loops, continue is useful for filtering out specific values or conditions without breaking the entire loop structure.

Syntax

```
for item in sequence1:
    for item in sequence2:
        continue
```

Examples

```
# A nested for loop with an innermost continue statement

myFirstList = [1, 2, 3]
mySecondList = [1, 2, 3]

for i in myFirstList:
    for j in mySecondList:
        if j == 3:
            continue  # Skip the iteration when j equals 3
        print(i, j, end=" ")
```

```
# A nested for loop with an outer continue statement

myFirstList = [1, 2, 3]
mySecondList = [1, 2, 3]

for i in myFirstList:
    if i == 2:
        continue  # Skip the iteration when i equals 2
    for j in mySecondList:
        print(i, j, end=" ")
```

Nested for loop with pass statements

Description

A **for loop** with a pass statement is used when a syntactically required block of code is needed, but no action should be performed at that point in the loop. The **pass statement acts as a placeholder**, **doing nothing when executed**, and is often used during development as a temporary stub or to handle a condition without affecting the loop's logic. Using pass allows the code to run without errors while signaling that a block is intentionally left empty for future implementation.

Syntax

```
See example below
```

Examples

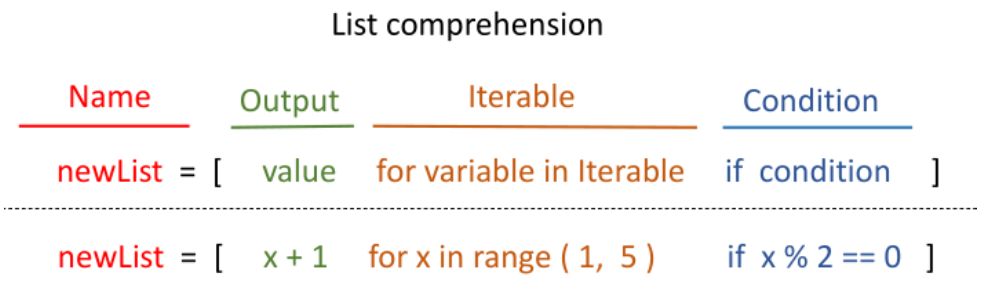
```
# Nested for loop with an outermost pass statement

myFirstList  = [1, 2, 3]
mySecondList = [1, 2, 3]
```

```
for i in myFirstList:
    if i == 2:
        pass  # Placeholder, does nothing for i = 2
    for j in mySecondList:
        print(i, j, end=" ")
```

For loops using list comprehension

Description

A **for loop** can be concisely expressed using list comprehension, which provides a readable and efficient way to create new lists by applying an expression to each item in a sequence. List comprehensions are often cleaner and faster than traditional for loops for simple operations. They can also include nested loops or multiple conditions, making them a powerful tool for concise and expressive list generation.



Syntax

```
    [expression for item in iterable if condition]
```

Examples

```
# List comprehension to create a list of numbers

myNumberList = [myNum for myNum in range(1, 6)]

print(myNumberList)
```

```
# Regular for loop, equivalent to the list comprehension

myNumberList = []

for myNum in range(1, 6):
    myNumberList.append(myNum)

print(myNumberList)
```

```
# List comprehension with arithmetic and condition

myNumberList = [myNum + 10 for myNum in range(0, 10) if myNum % 2 == 0]

print(myNumberList)
```

```
# Regular for loop, equivalent to the list comprehension

myNumberList = []

for myNum in range(0, 10):
    if myNum % 2 == 0:
```

```
            myNumberList.append(myNum + 10)

    print(myNumberList)
```

List comprehension with nested for loops

Description

Nested for loops can also be used within **list comprehensions** to create complex lists by **combining multiple sequences**. This allows you to iterate over multiple levels of data in a single line of code. **List comprehensions with nested loops are especially useful for flattening matrices**, generating Cartesian products, or performing operations across multiple datasets in a compact and readable format. However, readability can decrease as complexity increases.

Syntax

```
[expression for item1 in sequence1 for item2 in sequence2]
```

Examples

```
# List comprehension with nested for loops

myNumberList = [n for m in range(0, 3) for n in range(0, 3)]

print(myNumberList)
```

```
# Regular for loop, equivalent to the nested list comprehension

myNumberList = []

for m in range(0, 3):
    for n in range(0, 3):
        myNumberList.append(n)

print(myNumberList)
```

```
# List comprehension with nested for loops: m x n matrix

myMatrixList = [[m for m in range(0, 3)] for n in range(0, 3)]

print(myMatrixList)
```

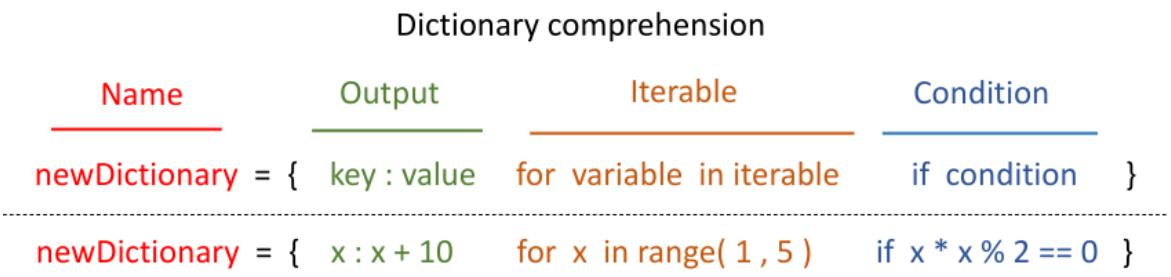
```
# Regular for loop, equivalent to the nested list comprehension: m x n matrix

myMatrixList = []

for m in range(0, 3):
    myTempList = []
    for n in range(0, 3):
        myTempList.append(n)
    myMatrixList.append(myTempList)

print(myMatrixList)
```

For loops using dictionary comprehension

Description

Dictionary comprehension is a concise and readable way to create dictionaries quickly. It can also serve as a data filter, making it easier to read and understand than using traditional loops. The syntax is similar to list comprehension, and you can include an if statement to filter

which items are added to the dictionary. Dictionary comprehensions are particularly useful for transforming data, filtering keys or values, and efficiently creating mappings.



Syntax

```
    {key:value for item in iterable if condition}
```

Examples

```
# Dictionary comprehension with a condition

newDictionary = {myNum: myNum + 10 for myNum in range(1, 15) if myNum * myNum % 2 == 0}

print(newDictionary)
```

```
# Regular for loop, equivalent to the dictionary comprehension

newDictionary = {}

for myNum in range(1, 15):
    if myNum * myNum % 2 == 0:
        newDictionary[myNum] = myNum + 10

print(newDictionary)
```

For loops using set comprehension

Description

Set comprehension is a concise way to create sets using a for loop in a single line of code. It uses **curly braces {} ** and **automatically removes duplicate values**, since sets store only unique elements. Set comprehension is particularly **useful for filtering unique results** from a sequence or applying transformations while eliminating duplicates.

Syntax

```
{expression for item in iterable if condition}
```

Examples

```
# Set comprehension in Python

mySetList = {myNum + 2 for myNum in range(1, 6)}
print(mySetList)
```

```
# Regular for loop, equivalent to the set comprehension

newSet = set()

for myNum in range(1, 6):
    newSet.add(myNum + 2)

print(newSet)
```

For loops using tuple comprehension

Description

There is no tuple comprehension in Python. What may appear to be a tuple comprehension—using parentheses () — is actually a generator expression, not a tuple. A generator expression creates an iterator that yields items one by one and does not store all elements in memory at once, unlike a tuple. This makes generator expressions memory-efficient for processing large sequences.

Syntax

```
    See examples below
```

Examples

```
# Generator expression (not a tuple) in Python

myTupleList = (myNum for myNum in range(1, 6))
print(myTupleList)
```

```
# Mimicking tuple comprehension using a generator

myTupleList = tuple(myNum + 2 for myNum in range(1, 6))
print(myTupleList)
```

For loop using the built-in functions

Description

For loops can be combined with **built-in functions to process and manipulate sequences** in a clear and efficient way. Commonly used functions include *range()*, *enumerate()*, *zip()*, *sorted()*, and *reversed()*. The *range()* function generates a sequence of numbers, *enumerate()* allows you to loop over a sequence while accessing both the index and the value, and *zip()* combines two or more iterables into pairs. Using these functions with for loops makes them more versatile and powerful for data processing and iteration tasks.

Syntax

```
    See examples below
```

Examples

```
# For loop using len() and range() functions

myNumberList = [1, 2, 3, 4, 5]

for index in range(len(myNumberList)):
    print(myNumberList[index], end=" ")
```

```
# For loop using the enumerate() function

myNumberList = [1, 2, 3, 4, 5]

for index, value in enumerate(myNumberList):
    print((index, value), end=" ")
```

```
# For loop with zip() function example using a fish list

myNumberList = [1, 2, 3, 4, 5]
myFishList   = ["Bass", "Red Snapper", "Karp", "Pike", "Perch"]

for number, fish in zip(myNumberList, myFishList):
    print(number, fish)
```

4.3 While loops

Description

A **while loop** is used to repeatedly execute a block of code as long as a specified condition is True. Unlike a for loop, which iterates over a known sequence, **a while loop is condition-controlled and continues running until the condition becomes False**. If the condition never becomes False, the loop can become infinite, so it is important to update the loop variable appropriately within the loop body. While loops are particularly useful when the number of iterations is unknown in advance and the loop should continue until a specific state or condition is reached.

Syntax

```
while condition: block of code
```

Examples

```
# While loop with pass statement

while False: pass
```

```
# While loop with pass statement

while 0: pass
```

```
# Infinite while loop with pass statement

while 1: pass
```

```
# Infinite while loop with print statement

while True: print("Infinite loop")
```

```
# While loop with comparison operator (<)

myFirstVariable = 5
i = 0

while i < myFirstVariable:
    print(i, end=" ")
    i += 1
```

```
# While loop with comparison operator (<=)

myFirstVariable = 5
i = 0

while i <= myFirstVariable:
```

```
    print(i, end=" ")
    i += 1
```

While loop with the break, continue, and pass statements

Description

while loops can be enhanced using the **break**, **continue**, and **pass** statements to control the flow of execution. The break statement immediately exits the loop when a specified condition is met, stopping further iterations. The continue statement skips the rest of the current iteration and moves to the next one, which is useful for ignoring specific conditions without ending the loop. The pass statement does nothing when executed and serves as a placeholder when a syntactically valid block of code is required but no action is needed.

These control statements make while loops more flexible and allow for precise control over loop behavior.

Syntax

```
while condition:
    if condition:
        break

while condition:
    if condition:
        continue

while condition:
    if condition:
        pass
```

Examples

```
# While loop with comparison (<) and break statement

myFirstVariable = 5
i = 0

while i < myFirstVariable:
    print(i)
    if i == 3:
        break  # stops the loop even though the while condition is still True
    i += 1
```

```
# While loop with comparison (<) and continue statement

myFirstVariable = 5
i = 0

while i < myFirstVariable:
    i += 1
    if i == 3:
        continue  # skip the rest of this iteration
    print(i)
```

```
# While loop with comparison (<) and pass statement

myFirstVariable = 5
i = 0

while i < myFirstVariable:
    i += 1
    if i == 3:
        pass  # placeholder, does nothing
    print(i)
```

While loop with else statement

Description

A while loop can include an optional else clause, which executes after the loop finishes normally. This feature is useful for performing actions once the loop condition becomes False, allowing post-loop processing or providing confirmation that the loop ran to completion.

Syntax

```
while condition:
    block of code
else:
    block of code
```

Examples

```
# While loop with else statement

myFirstVariable = 5
i = 0

while i < myFirstVariable:
    print(i)
    i += 1
else:
    print("i is not less than 5 anymore")
```

Nested While loops

Description

A **nested while loop** is a **while loop placed inside another while loop**, allowing repeated iterations within repeated iterations. This is useful for working with multi-dimensional data or performing repetitive checks inside a broader condition. For example, the outer loop could control rows and the inner loop could handle columns in a matrix. Each time the outer loop executes once, the inner loop runs completely if its condition remains True. It is important to update the loop variables properly in both loops to avoid infinite loops.

Syntax

```
while condition1:
    block fo code
        while condition2:
            block of code
```

Examples

```
# Nested while loops

myFirstVariable = 5
i, j = 0, 0

while i < myFirstVariable:
    print(f"Outer loop i = {i}")
    i += 1
    while j < myFirstVariable:
        print(f"  Inner loop j = {j}")
        j += 1
```

Nested While loop with break statement

Description

A **nested loop** with a **break statement** allows you to exit a loop based on specific conditions. When a break is encountered inside the inner loop, it immediately terminates that loop, stopping further iterations of the inner loop. A break in the outer loop will exit the outer loop entirely once its condition is met. This provides precise control over both inner and outer loop execution.

Syntax

```
   while condition1:
       while condition2:
           if condition:
               break
       else:
           continue
       break
```

Examples

```
# Nested while loops with break statements

myFirstVariable = 5
i = 0

while i < myFirstVariable:
    j = 0  # reset inner loop counter for each outer iteration
    print(f"Outer loop i = {i}")

    while j < myFirstVariable:
        if j == 2:
            print("  Breaking inner loop when j = 2")
            break  # exits inner loop
        print(f"  Inner loop j = {j}")
        j += 1

    i += 1
else:
    print("i is no longer less than 5")
```

Nested While loop with continue statement

Description

A **while loop** with a continue statement allows you to skip the rest of the current iteration and immediately return to the loop's condition check. This is useful for ignoring specific cases without exiting the loop, enabling the loop to continue processing remaining iterations.

Syntax

```
   while condition1:
       while condition2:
           if condition:
               continue
```

Examples

```
# Nested while loops with continue statement

i = 0
outerLimit = 3
innerLimit = 3

while i < outerLimit:
    j = 0  # reset inner loop counter for each outer iteration
    print(f"Outer loop i = {i}")
    i += 1

    while j < innerLimit:
        j += 1
        if j == 2:
            print("  Skipping iteration when j = 2")
            continue  # skip printing when j == 2
        print(f"  Inner loop j = {j}")
else:
    print("i is no longer less than 3")
```

Nested While loop with pass statement

Description

A **nested while loop** with the pass statement allows you to structure complex logic while leaving placeholder code for sections that are not yet implemented. The pass statement does nothing when executed; it simply acts as a syntactic placeholder where code is required but not yet written.

Syntax

```
   while condition1:
       while condition2:
           if condition
               pass
```

Examples

```
# Nested while loops with else statements

i, j = 0, 0
outerLimit, innerLimit = 5, 5

while i < outerLimit:
    print(f"Outer loop: i = {i}, j = {j}")
    i += 1
    j = 0  # reset inner loop counter for each outer iteration

    while j < innerLimit:
        print(f"  Inner loop: i = {i}, j = {j}")
        j += 1
    else:
        pass  # placeholder: can add post-inner-loop logic here
else:
    print("i is no longer less than 5")
```

Section 5: Functions

5.1 User defined functions

5.2 Global and local variables

5.3 Parameters vs arguments

5.4 Recursive functions

5.5 Lambda functions

5.6 Function annotations

5.1 User-defined functions

Description

A function is a block of organized code designed to perform a specific task. Python provides many built-in functions such as print(), len(), and range(). Additionally, you can create your own user-defined functions to improve the readability, reusability, and organization of your code.

Python executes programs from top to bottom, so a function must be defined before it is called. The general rules for defining a function are:

1. Use the def keyword followed by the function name and parentheses, ending with a colon (:).
2. Optionally, include parameters inside the parentheses.
3. Indent the code block containing the statements that the function should execute.
4. Use the return statement to exit the function and optionally pass a result back to the caller.

Syntax

```
    def functionName(parameters): block of code / return value
```

Basics of user-defined functions

Description

User-defined functions allow programmers to create reusable blocks of code that perform specific tasks. A function is defined using the def keyword, followed by the function name, parentheses (which may include parameters), and a colon. The function body is indented and contains the code to be executed when the function is called. Functions can return values using the return statement, or simply perform actions without returning anything. User-defined functions improve code modularity, readability, and maintainability by avoiding repetition and organizing logic into clear, logical units.

Syntax

```
    See examples below
```

Examples

```
# Defining an empty function

def myEmptyFunction(): # Function definition
    pass
```

```
# Defining a function

def myFirstFunction():
    pass               # Body statement
```

```
# Defining a function and making a call

def myFirstFunction():
    pass

myFirstFunction()        # Function call
```

```
# Defining function with the body statement making a call

def myFirstFunction():    # Function definition
  print("Hello World!")   # Body statement

myFirstFunction()         # Function call
```

Function with the return statement

Description

A function with a return statement sends a value back to the part of the program where the function was called. The return statement terminates the function execution and returns a value to the caller. The returned value can be of any data type, including int, float, complex data types, sequences, or user-defined objects such as classes and functions. The return value can be omitted, or a function can have a single return statement. A function may include multiple return statements, often within conditional blocks, but only the first one reached during execution will be executed, and the function will exit immediately. If no return statement is provided, the function returns None by default. Using return allows functions to produce outputs that can be reused elsewhere in a program, making them more versatile and powerful.

Syntax

```
    See examples below
```

Examples

```
# Defining function with a return statement

def myTestFunction(): return

myTestFunction()
```

```
# Defining function with an explicit return statement

def myFirstFunction():
    print("Before the return statement")  # This line runs
    return                                # Function exits here
    print("After the return statement")    # This line is skipped

myFirstFunction()
```

```
# Defining function with an explicit return value in one line

def myFirstFunction(): return 2 + 3

print(myFirstFunction())  # Output: 5
```

```
# Function that returns a list of even numbers using list comprehension

def myListFunction(myNumberList):
```

```
    # Create a new list with only even numbers
    evenNumbers = [i for i in myNumberList if i % 2 == 0]
    return evenNumbers

myNumberList = [1, 2, 3, 4, 5, 6, 7, 8]

print(myListFunction(myNumberList))
```

5.2 Global and local variables

Description

Local and **global** variables **determine the scope and accessibility of data within a Python program**. A local variable is defined inside a function and can only be accessed within that function. In contrast, a global variable is defined outside any function and can be accessed throughout the entire program, including inside functions. To modify a global variable from within a function, you must use the global keyword; otherwise, Python treats any assignment as creating a new local variable.

Syntax

```
Local variable

def functionName():
    localVariable = value

Global variable

def functionName():
    global globalVariable
```

Examples

```
# Global variable is defined outside of a function

myGlobalVariable = "cool!"

def myTestFunc():
  print("Python is " + myGlobalVariable)

myTestFunc()
```

```
# Local variable is defined inside of a function

def myTestFunc():
  myLocalVariable = "cool!"
  print("Python is " + myLocalVariable)

myTestFunc()
```

```
# Example: Global vs Local Variables

# Global variable (accessible anywhere in the program)
myGlobalVariable = "awesome"

def myTestFunc():
    # Local variable (only accessible inside this function)
    myLocalVariable = "fantastic"

    # Accessing the global variable inside the function
    print("Python is " + myGlobalVariable + "! - from function")

    # Accessing the local variable inside the function
    print("Python is " + myLocalVariable + "! - from function")
```

```
# Call the function
myTestFunc()

# Uncommenting the line below will cause an error because myLocalVariable is local to myTestFunc
# print("Python is " + myLocalVariable)
```

```
# Example: Local variable shadows global variable

# Global variable
myVariable = "awesome"

def myTestFunc():
    # Local variable with the same name as the global variable
    myVariable = "fantastic"

    # Inside the function, Python uses the local variable
    print("Python is " + myVariable + "! - from function")

# Call the function
myTestFunc()

# Outside the function, the global variable is used
print("Python is " + myVariable + "! - from global scope")
```

```
# Example: Creating a global variable inside a function

# Global variable defined outside the function
myGlobalVariable = "awesome"

def myTestFunc():
    # Declare a variable as global to make it accessible outside the function
    global myLocalVariable
    myLocalVariable = "fantastic"

    # Accessing both global variables inside the function
    print("Python is " + myGlobalVariable + "! - from function")
    print("Python is " + myLocalVariable + "! - from function")

# Call the function
myTestFunc()

# The variable created inside the function is now global
print("Python is " + myLocalVariable + "! - from global scope")
```

```
# Example: Modifying a global variable inside a function

# Global variable defined outside the function
myVariable = "awesome"

def myTestFunc():
    global myVariable  # Declare we want to modify the global variable
    myVariable += " fantastic"  # Update the value of the global variable
    print("Python is " + myVariable + "! - from the function")

# Call the function
myTestFunc()

# The global variable has been modified
print("Python is " + myVariable + "! - from global scope")
```

5.3 Parameters vs arguments

Description

Function Parameters and Arguments

Parameter: A variable listed inside the parentheses when a function is defined. It is local to the function and can only be accessed inside the function.

Argument: The actual value you pass to a function when calling it. Arguments can be values, variables, or expressions.

Shorthand notations:

*args → allows passing a variable number of positional arguments.

**kwargs → allows passing a variable number of keyword arguments.

Ways to call a function:

Positional arguments: Values are assigned to parameters in order.

Keyword arguments: Values are assigned to parameters using their names.

Mixed arguments: A combination of positional and keyword arguments.

Important rule: The number of arguments provided must match the number of parameters defined (unless using *args or **kwargs).

Syntax

```
def  functionName(param 1, param 2, ..., param n): block of code
call functionName(arg 1, arg 2, ..., arg n)

def  functionName(param 1, param 2, ..., param n): block of code
call functionName(arg 1 = val 1, arg 2 = val 2, ..., arg n = val n)

def  functionName(param 1, param 2, ..., param n): block of code
call functionName(arg 1, arg 2 = val 2, ..., arg n = val n)
```

Positional arguments

Description

Positional arguments are the most common way to pass values to a function.

Each argument is assigned to a function parameter based strictly on its position in the function call.

The first argument matches the first parameter,

The second argument matches the second parameter, and so on.

The order of arguments is important; changing the order can lead to unexpected results.

Syntax

```
See examples below
```

Examples

```
# Function with one positional argument

def catchFish(fishName):
    print("Caught a", fishName)

catchFish("Bass")
catchFish("Trout")
catchFish("Salmon")
```

```
# Function with two positional arguments

def fishCatch(fishType, quantity):
    print("Caught", quantity, fishType + "(s)")

fishCatch("Bass", 3)
fishCatch("Trout", 5)
```

```
# Function with three positional arguments (fishing-related)

def fishCatch(fishType, quantity, location):
    print("Caught", quantity, fishType + "(s) at", location)

fishCatch("Bass", 3, "Lake Tahoe")
fishCatch("Trout", 5, "River Nile")
```

Keyword arguments

Description

Keyword arguments allow you to pass values to a function by explicitly specifying the parameter names in the function call, rather than relying on their positions. This improves code readability and flexibility, especially for functions with multiple parameters. Keyword arguments also let you skip parameters that have default values.

Syntax

```
See examples below
```

Examples

```
# Function with keyword arguments

def fishCatch(fishType, quantity, location):
    print("Caught", quantity, fishType + "(s) at", location)

# Using keyword arguments
fishCatch(fishType="Bass", quantity=3, location="Lake Tahoe")
fishCatch(location="River Nile", quantity=5, fishType="Trout")
```

```
# Function with various key=value arguments
# Keyword arguments allow calling parameters in any order

def catchFish(fishType, quantity, location):
    print("Caught", quantity, fishType + "(s) at", location)

# Calling function with keyword arguments in any order
catchFish(location="Lake Tahoe", fishType="Bass", quantity=3)
catchFish(quantity=2, location="River Nile", fishType="Trout")
```

Mixed arguments - a mix of positional and keyword arguments

Description

Mixed arguments refer to using a combination of positional and keyword arguments in a function call. When using mixed arguments, all positional arguments must appear first, followed by keyword arguments. This approach provides flexibility by allowing required arguments to be passed in order while optional arguments can be specified by name. Using mixed arguments improves readability and helps avoid errors, especially in functions with many parameters, some of which may have default values.

Rule of thumb: Positional arguments must always come before keyword arguments.

Syntax

```
See examples below
```

Examples

```
# Function with both positional and keyword arguments

def fishInfo(name, species, length):
    print("Fish:", name, "| Species:", species, "| Length:", length, "cm")

# Call the function with one positional and two keyword arguments
fishInfo("Nemo", species="Clownfish", length=15)
```

```
# Fish-themed function with positional and keyword arguments

def fishInfo(species, length, weight):
    print(f"Fish: {species}, Length: {length} cm, Weight: {weight} kg")

# Positional and keyword arguments
fishInfo("Salmon", length=75, weight=4.5)
#fishInfo("Bass", 23, length=50)
```

Functions with arbitrary number of arguments

Description

The *args syntax in Python allows a function to accept an unknown number of positional arguments. The asterisk * before args collects all extra positional arguments passed to the function into a tuple, which can then be iterated over inside the function. This is useful when you don't know in advance how many arguments a function might receive.

Syntax

```
def functionName(*args):
```

Examples

```
# Arbitrary number of arguments (*args), receives a tuple of fish names

def showFish(*args):
    print(args)

showFish("Salmon", "Bass", "Trout", "Pike")
```

Functions with arbitrary number of keyword positional arguments

Description

To accept an arbitrary number of keyword arguments in a function, you use the **kwargs syntax. The double asterisk (**) indicates that the function can receive multiple keyword arguments, which are collected into a dictionary inside the function.

Syntax

```
def functionName(**kwargs):
```

Examples

```
# Arbitrary number of keyword arguments (**kwargs).
# The **kwargs parameter collects all passed keyword arguments into a dictionary.

def showFishDetails(**kwargs):
    print(kwargs)
```

```
showFishDetails(Salmon=5, Bass=3, Trout=8, Pike=2)
```

```
# Arbitrary number of keyword arguments for fish with different keys

def myFishFunction(**kwargs):
    print(kwargs)

myFishFunction(favorite="Salmon", rare="Sturgeon", common="Perch")
```

Functions with arbitrary number of positional arguments and keyword arguments

Description

It is possible to create a function that accepts both an arbitrary number of positional arguments (*args) and an arbitrary number of keyword arguments (**kwargs). The positional arguments are collected into a tuple, while the keyword arguments are collected into a dictionary.

Rule of thumb: *args must always come before **kwargs in the function definition.

Note

*args collects all positional arguments passed to the function into a tuple.

**kwargs collects all the keyword arguments to the function into a dictionary.

Syntax

```
def functionName(*args, **kwargs):
```

Examples

```
def myFishFunction(*args, **kwargs):
    print("Positional arguments (fish names):", args)
    print("Keyword arguments (fish types):", kwargs)

myFishFunction("Salmon", "Trout", favorite="Sturgeon", rare="Perch")
```

Functions with default parameters

Description

Functions with default parameters allow you to assign a default value to one or more parameters when defining a function. If no argument is provided for that parameter during the function call, the default value is used. Default parameters are specified using the assignment operator (=) in the function definition. Important: all default parameters should come after required (non-default) parameters in the function signature to avoid syntax errors.

Syntax

```
def  functionName(param 1, param 2 = val 2, ..., param n = value n): block of code
call functionName(arg 1, arg 2, ..., arg n)
```

Examples

```
def myRegularFunction(myVar1, myVar2=0.34, myVar3=10):
    print(myVar1, myVar2, myVar3)
```

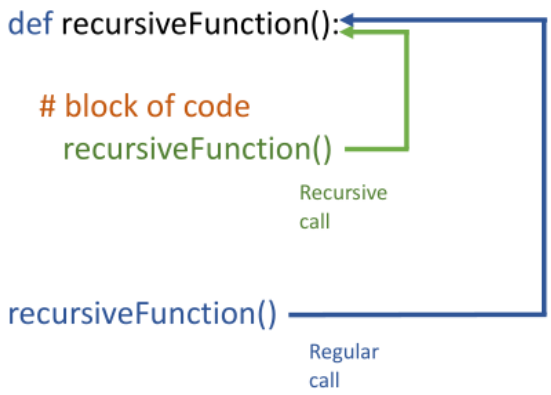
```
# Only the first argument is provided, others use default values
myRegularFunction(0.1)

# First and second arguments provided, third uses default
myRegularFunction(0.1, 0.22)

# All arguments provided, defaults are overridden
myRegularFunction(0.1, 0.22, 123456)
```

5.4 Recursive functions

Description

A recursive function is a function that calls itself to solve a problem. This technique is useful for breaking complex problems into simpler problems of the same type. A recursive function must have a base case to stop the recursion and prevent an infinite loop. The recursive case reduces the problem and calls the function again. Recursive functions are powerful for tasks with repeated patterns, such as traversing trees or implementing divide-and-conquer algorithms.

In short, a recursive function is a function that refers to itself.



Syntax

```
def functionName(parameters):
    if condition:
        return value
    else
        # recursive case
    return functionName(updated parameters)
```

Examples

```
# Recursive function to calculate factorial (5! = 1*2*3*4*5)

def myRecursiveFunction(x):
    """Recursive function to find factorial of x"""
    if x == 1:
        return 1
    else:
        return x * myRecursiveFunction(x - 1)

x = 5
print("The factorial of", x, "is", myRecursiveFunction(x))
```

Functions with data sequences passed as arguments

Description

Functions can accept data sequences such as strings, lists, tuples, and sets as arguments, allowing them to perform operations on collections of values. When a sequence is passed to a function, it can be accessed using loops or sequence operations like indexing, slicing, and built-in functions. Creating functions that accept sequences enables you to manipulate and process the data directly within the function. Below are examples using a list, dictionary, tuple, and set.

Syntax

```
def functionName(sequence):
    for item in sequence:
        block of code
```

Examples

```
# Function that accepts a list as an argument

def myFishFunction(fishList):
    for fish in fishList:
        print(fish, end="; ")

fishList = ["Carp", "Bass", "Red Snapper", "Labeo culbasu", "Chanda Nama", "Ruffe"]

myFishFunction(fishList)
```

```
# Function with a tuple as an argument
def myFishFunction(fishTuple):
    for fish in fishTuple:
        print(fish, end="; ")

fishTuple = ("Carp", "Bass", "Red Snapper", "Labeo culbasu", "Chanda Nama", "Ruffe")

myFishFunction(fishTuple)
```

```
# Function with a dictionary to show values and items
def myFishFunction(fishDict):
    for key, value in fishDict.items():
        print(f"{key}: {value}", end="; ")

fishDict = {
    "Fish1": "Carp",
    "Fish2": "Bass",
    "Fish3": "Red Snapper",
    "Fish4": "Labeo culbasu",
    "Fish5": "Chanda Nama",
    "Fish6": "Ruffe"
}

myFishFunction(fishDict)
```

```
# Functions with a set as an argument

def myFishFunction(fish):
    for i in fish:
        print(i, end="; ")

fish = { "Carp", "Bass", "Red Snapper", "Labeo culbasu", "Chanda Nama", "Ruffe" }

myFishFunction(fish)
```

Modyfing a list inside of the function

Description

You can return a modified sequence from a function just like any other value. After processing or transforming the sequence, the function can use the return statement to send the modified results back to the caller.

The data sequence can be modified inside of the function!

Syntax

```
def functionName(sequence):
    block of code
    modified_sequence = ...
    return modified_sequence
```

Examples

```
# Function to modify a fish list

def modifyFishList(fishList):
    del fishList[2]  # Deleting the fish at index 2
    return fishList

myFishList = ["Carp", "Bass", "Red Snapper", "Labeo Culbasu", "Chanda Nama"]

print("Fish list before function call:", myFishList)

modifyFishList(myFishList)
print("Fish list after function call:", myFishList)
```

```
# Function to add a fish to the list

def addFish(fishList, newFish):
    fishList.append(newFish)
    print("Fish list inside function:", fishList)

myFishList = ["Carp", "Bass", "Red Snapper"]

print("Fish list before function call:", myFishList)

addFish(myFishList, "Labeo Culbasu")
print("Fish list after function call:", myFishList)
```

5.5 Lambda function

Description

A lambda function is a small, anonymous function defined using the lambda keyword instead of def. It is useful for short, simple operations where a full function would be too verbose. A lambda function can take any number of arguments but can only contain one expression—no multiple statements or assignments. Lambda functions are often used as arguments to functions like *map()*, *filter()*, or *sorted()* for inline, on-the-fly operations.

In short: anonymous, concise, single-expression functions.

Syntax

```
lambda variable: one expression
```

Examples

```
# A lambda function to increase the length of a fish by 2 units
# lambda arguments : expression

increaseFishLength = lambda fishLength: fishLength + 2
print("New fish length:", increaseFishLength(5))
```

```
# A lambda function calculates total weight of fish given quantity and weight per fish

calculateFishWeight = lambda quantity, weightPerFish: quantity * weightPerFish
print("Total fish weight:", calculateFishWeight(10, 2.5))
```

```
# A lambda function to calculate average weight per fish batch

AverageFishWeight = lambda weightBatch1, weightBatch2, totalFish: (weightBatch1 + weightBatch2) / totalFish
print("Average fish weight:", AverageFishWeight(0.3, 0.4, 1.23))
```

5.6 Function annotations

Description

Function annotations are optional metadata for function parameters and return values. They provide hints about the expected types or purpose of parameters and what the function returns. An annotation is placed before a parameter's default value and is separated by a colon (:). You can also annotate *args and **kwargs parameters. Note that lambda functions do not support annotations.

Syntax

```
def functionName(param1: Type1, param2: Type2, ..., paramN: TypeN) -> returnType:
def functionName(*args: argType, **kwargs: kwargType) -> returnType:
```

Return value annotation

Description

The -> operator in a function definition is used for the return value annotation. Function annotations are optional and only serve as hints for readability, type checking, or documentation—they do not affect how Python runs the code. The interpreter ignores it.

Syntax

```
def functionName(parameter: type) -> returntype:
    block of code
    return value
```

Examples

```
# x: int  says that x is expected to be an integer
# -> float says the function is expected to return a float

def myFunction(x: int) -> float:
    return x / 2
```

```
# Return value annotation with integer type

def myIntFunction(myVal1: int, myVal2: int) -> int:
```

```
  return (myVal1 * myVal2)

print(myIntFunction(1,2))
```

```
# Return value annotation with float type

def myFloatFunction(myVal1: float, myVal2: float) -> float:
  return (myVal1 * myVal2)

print(myFloatFunction(1.1, 2.4))
```

```
# Return value annotation with str type

def myStrFunction(myVal1: str, myVal2: str) -> str:
  return (myVal1 + myVal2)

print(myStrFunction("***", "*****"))
```

```
# To annotate the list return type you can use the return value annotation.

def myListFunction() -> list[str]:
   myList=[1,2,3,4,5]
   return myList

myList=myListFunction()
print(myList)
```

```
# To annotate the dictioanry return type you can use the return value annotation.

def myDictFunction() -> dict[str:int]:
   myDict={"IDnumber":123456,"age":32}
   return myDict

myDict=myDictFunction()
print(myDict)
```

```
# Return value annotation with a default parameter, the default parameter will be ignored.

def myTestFunction(myVal1: int, myVal2: int = 5) -> int:
  return (myVal1 * myVal2)

print(myTestFunction(1,2))
```

```
# To annotate the tuple return type you can use the return value annotation.

def myTupleFunction(*args) -> tuple[int]:
    return args

myTuple = myTupleFunction(1,2,3,4,5)

print(myTuple)
print(type(myTuple))
```

```
# To annotate the tuple return type you can use the return value annotation.

def myDictFunction(**kwargs) -> dict[str,str]:
    return kwargs

myDict = myDictFunction(Name="Oleaf", age="39")

print(myDict)
print(type(myDict))
```

Section 6: Modules

6.1 Built-in modules

6.1 Built-in modules

Description

We decompose our code in smaller pieces to make our code more readable. The way we can do this is to import modules to the code. A file that contains a set of functions is called a module. This module can be accessed from a program using "**import**" keyword. A module can also contain variables of all data types. You need to import a module to access all the functions and variables. Most python distributions come with many build-in modules like os, sys, time, math ... Here are a few examples with built-in functions such as os, sys, time, datetime, and math.

Syntax

```
    import module
```

Examples

```
# To get current working directory path

import os

cwd = os.getcwd()

print("My current working directory: ", cwd)
```

```
# To list content of your directory

import os

path="/content"

dir_content=os.listdir(path)

print("Content of my directory: ", dir_content)
```

```
# In this example the version of Python interpreter is return as string
import sys

print(sys.version)
```

```
# In this example we use sys.path that returns a list of directories
# where the Python interpreter will search for the available modules.
import sys

print(sys.path)
```

```
# How to use time module
import time

print(time.strftime("%H:%M:%S"))
```

```
# Date time now
import datetime
```

```
# To specify exact entity we need to use the following syntax.
import math

print(math.pi)
```

Note

The asterisk (*) in *myFirstList is the unpacking operator. It tells Python to unpack the list and pass its elements as individual arguments to a function. myFirstList = [1, 2, 3]; print(*myFirstList) -> print(1,2,3) So instead of printing the whole list as a single object ([1, 2, 3]), it prints the individual elements separated by spaces: 1 2 3. ()* is the unpacking operator for lists, tuples, sets, etc. It splits a collection into individual arguments. Very useful in *print()* and function calls where you want to deal with elements separately.

6.2 Location of modules

Description

A standard module in Python is a built-in component of the Python Standard Library, which comes pre-installed with every Python distribution. These modules provide essential functionality such as mathematical operations **(math)**, file and directory management **(os)**, and system interaction **(sys)**. When you import a standard module using the import statement, Python searches for it following a specific order defined by the sys.path list. This list includes the **current working directory**, **standard library directories**, and **any additional paths specified by environment variables like PYTHONPATH**. Python locates modules by scanning these directories for files with extensions such as .py, .pyc, or platform-specific compiled modules. Since standard modules are part of Python's core installation, they are usually found immediately and require no additional setup or installation.

Syntax

```
import sys

print(sys.path)
```

Examples

```
# This prints the list of directories Python searches in order when you use import some_module.

import sys

print(sys.path)
```

```
# Find a location of an imported module

import os

print(os.__file__)
```

```
# To check my current path

import os

print(os.getcwd())
```

```
# To check absolute path

import os

print(os.path.abspath('content'))
```

```
# To append a location to the path, realtive or absolute path.

import sys

#sys.path.append('/path/to/your/directory')

sys.path.append('/content/content')

print(sys.path)
```

6.3 Importing modules

Description

Modules are imported using the *import* statement, which allows you to access functions, classes, and variables defined in another file or built-in library. You can import an entire module using *import module name* statement, access its members with dot notation. To simplify access or avoid naming conflicts, you can assign an alias as *import module name as alias*. If you need only specific componentsfrom a module, you can use *from module name import item1, item2, ... * which lets you to use those items directly withoutthe module prefix. Additionally, *from module name import * * will import all public components, but it can lead to the namespace conflicts.

Syntax

```
import module

from module import *

from module import entity1, entity2, ...

import module as short name or alias

from module import entity name as short name

Both statements ( 1. and 2. ) are importing all
the entities from the module. The difference is
that in the first example you need to use prefix
(module.method) to refer to the method and in the
second example you do not need to use the prefix.
```

Examples

```
# Importing just the entire module, importing all the entities.
import math

print(math.pi) # Built in
```

```
# You can import everything from the module using an asterisk.
# It is not recommended to do so since a lot of conflicts can appear with your
# own function names and they will be replaced. You also need to notice that
# you do not need to specify math prefix anymore. You can just use the name itself.

from math import *

print("Pi number", pi)
```

```
# Importing just some of the entities.
from math import acos, pi

print(acos(-1))
print(pi)
```

```
# You can also import a module using an alias to refer to the module,
# a shorter name or an acronym for example.
import math as mth

print("Pi number :", mth.pi)
```

```
# You can also import a module using an alias to refer to the module,
# a shorter name or an acronym for example.

import random as rnd

print(rnd.randint(0,9)) #Returns a number in a<=N<=b or [a,b] inclusive
```

```
# Generating an array of random numbers
from numpy.random import seed
from numpy.random import randint

#Every time you run the code, the same "random" numbers are generated.
seed(1)

randomValues = randint(0,10,20)
print(randomValues)
```

```
import sys

def exit():
  print('My own exit function!')

exit()

sys.exit()
```

```
from sys import exit

def exit():
  print('My own exit function!')

exit()
```

```
from sys import *

def exit():
  print('My own exit function!')

exit()
```

Note

When a module is imported in Python, its code is read and executed. All modules contain definitions for functions, classes, and variables. When you import a module, Python runs through the entire module, making its contents available to the importing file. Python also keeps track of all loaded modules and ensures that each module is initialized only once. You can have a module imported many times by other modules but it will be initialized only once.

Example: If you have myModule1.py is imported to myModule2.py, and then both are imported to main.py then myModule1.py will be initialized only once. Even though myModule1.py appears to be imported twice.

Python also assigns a special built-in variable, "_ _ *name* _ _", to every module. If the file is run directly (e.g., python myModule1.py), the "_ _ *name* _ _" is set to '_ _ *main* _ _'. If the file is imported as a module, "_ _ *name* _ _" is set to the module's name (usually the filename without the .py extension).

A module is initialized only once!

Syntax

```
if __name__ == "__main__":
  This block only runs when the file is executed directly
```

Examples

```
if __name__=='__main__':

   print('My module is run directly')
else:
   print('My module is run as imported')
```

Note

Most popular python modules and packages can be found at www.pypi.org

1. Data analysis

```
NumPy – module is used for variety of mathematical operations on arrays.

Pandas – module designed to work with the datasets, data analysis.
```

2. Artificial intelligence and machine learning

```
Scikit-learn – module with machine learning libraries.

TensorFlow – end-to-end machine learning module.

PyTorch – module with the largest machine learning library.

Keras – module with machine learning libraries.
```

3. Web development

```
Django – module designed to help build webpages using pyhton.

Flask  – a lightweight and flexible micro-framework.
```

4. Data visualization

```
MatplotLib – module for creating static, animated, and interactive graphs.
```

5. Scientific computing

```
SciPy – module with a lot of mathematical algorithms and functions.
```

Note

You can list all the classes and functions available in any Python library using the built-in **dir()** function. The **dir()** function returns all attributes of a **module**, including **functions**, **classes**, **constants**, and **variables**.

```
# Use the built-in dir() function
import numpy as np
print(dir(np))
```

```
# There is a dir() function that lists all the functions inside of a module

import numpy

myFirstList=dir(numpy)
print(*myFirstList,sep='|')
```

```
False_|ScalarType|True_|_CopyMode|_NoValue|__NUMPY_SETUP__|__all__|__array_api_version__|__builtins__|__cached__|__
```

```
# Use the built-in help() function
import numpy as np
help(np.mean)
```

```
# View the function's docstring directly
print(np.mean.__doc__)
```

6.4 User defined modules

Description

The following example should be performed in Jupyter Lab. Once you import a module the Python executes it immediately, that is why most of the modules contain functions and variables' definitions only. Some modules not only contain the functions but also some automated tests making sure that the functions work correctly. These tests are not activated upon the call but independently. Each module will be initialized only once eventhough you might have a few initializations in different files. Python will be aware of all the modules imported in other files.

Syntax

```
Creating a module

# mymodule.py

def myFunction(parameters):
    block of code
    return result

Using the module

import mymodule
mymodule.myFunction(arguments)
```

Examples

```
# myOwnModule
if __name__=="__main__":
  print("This module runs directly")
else:
  print("This module is imported")
```

```
#Main code
import myOwnModule
```

```
# Save your file as myFirstModule.py

def greeting(name):
  print("Hello, " + name)

worker={"name":"John", "age":48, "country":"Germany"}
```

```
# Import my own module
import myFirstModule

myFirstModule.greeting("Daniel")
myFirstModule.worker["age"]
```

6.5 Plotting module

Description

The plotiing module usually refers to the plotting libraries like Matplotlib or Seaborn, which are used to create visualizations such as line graphs, bar charts, histograms, scatterplots, animations, and more. Matplotlib is a versatile plotting library that provides a wide range of static, animated, and interactive plots. It allows a user to customize almost all aspects of a plot, including labels, titles, colors, and styles. It provides a higher level interface for creating aesthetically pleasing and informative statistical graphics, with buil-in themes and color pallets. This library is commonly used in data analysis, data science, and machine learning projects to visualize trends, distributions, and relationships between variables, helping to understand data better and communicate more effectively.

Matplotlib is the most popular plotting module. It is used to create static, animated, and interactive graphs!

Syntax

```
import matplotlib.pyPlot as plt
```

Examples

```
import matplotlib

print(dir(matplotlib))
```

```
# Importing the required module as alias plt
import matplotlib.pyplot as plt

# Generate x,y-axis values
x=[1,2,3,4]
y=[1,2,3,4]

# Plotting the points
plt.plot(x,y)

# Figure and axis titles
plt.title ('Figure Title', fontsize=12, color='red')
plt.xlabel('X-axis Title', fontsize=12, color='blue')
plt.ylabel('Y-axis Title', fontsize=12, color='black')

# Show the plot using show() function
plt.show()
```

```
# Importing the required module as alias plt and np
import matplotlib.pyplot as plt
import numpy as np

# Generate x,y-axis values
x = np.array([1, 8])
y = np.array([3, 10])

# Plotting the points
plt.plot(x, y)

# Figure and axis titles
plt.title ('Figure Title', fontsize=12, color='red')
plt.xlabel('X-axis Title', fontsize=12, color='blue')
plt.ylabel('Y-axis Title', fontsize=12, color='black')

# Show the plot using show() function
plt.show()
```

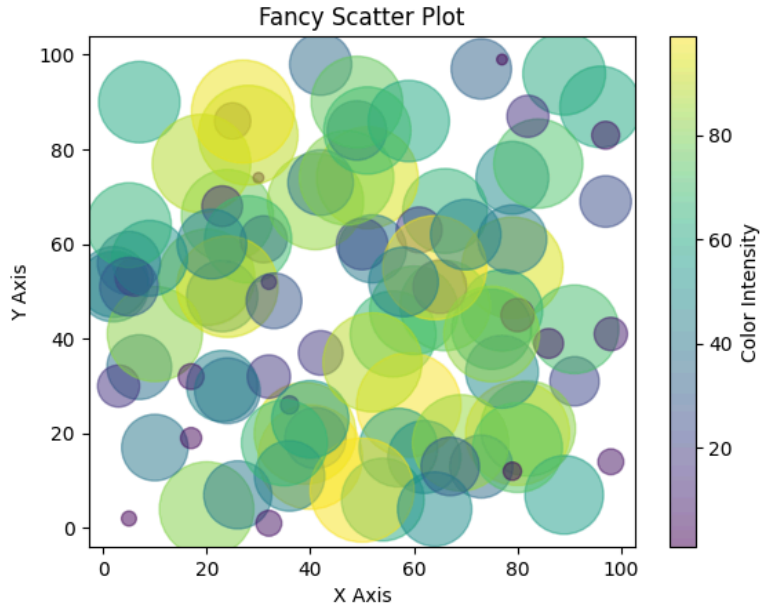
```
# Importing the required module as alias plt and np
import matplotlib.pyplot as plt
import numpy as np
```

```
# Generate 150 evenly spaced theta values from 0 to 2pi.
theta = np.linspace( 0 , 2 * np.pi , 150 )

# Setting up radius of the circle
radius = 0.2

# Calculating the x and y coordinates of points on a circle given the radius
x = radius * np.cos( theta )
y = radius * np.sin( theta )

# Creating two variables figure and axes
# figure represents the entire figure
# axes represent the plot itself
# plt.subplots(1) creates just one plot
figure, axes = plt.subplots( 1 )

# Plotting the points
axes.plot( x, y )

# Setting the aspect ratio to 1, plot will have equal scaling in x and y
axes.set_aspect( 1 )

# Show the title of the plot
plt.title( 'Parametric Equation Circle', fontsize=12, color='red' )
plt.xlabel('X-axis Title', fontsize=12, color='blue')
plt.ylabel('Y-axis Title', fontsize=12, color='black')

# Show the plot using show() function
```

```
# Importing the required module as alias plt, mth, and np
import matplotlib.pyplot as plt
import math as mth
import numpy as np

# Generate x from 0 to 1*pi (not included) with the step size of 0.01 between each value
x = np.arange(0,10 * np.pi, 0.01)
y = np.cos(x)

# Plotting the points
plt.plot(x, y)
plt.grid(True)

plt.xlabel('x', fontsize=12, color='blue')
plt.ylabel('cos(x)', fontsize=12, color='blue')
plt.title('Cosine Function', fontsize=12, color='blue')

# Show the plot using show() function
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt

# print(plt.colormaps())

x = np.random.randint(0,100,100)
y = np.random.randint(0,100,100)

scatter = np.random.randint(0,100,100)

plt.title('Fancy Scatter Plot')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

plt.scatter(x=x, y=y, c=scatter, s=scatter*30, cmap='viridis', alpha=0.5)
plt.colorbar(label='Color Intensity')

plt.show()
```

```
['magma', 'inferno', 'plasma', 'viridis', 'cividis', 'twilight', 'twilight_shifted', 'turbo', 'berlin', 'managua',
```



6.6 Molecular visualization modules

Description

Py3Dmol and RDKit are popular Python modules widely used for molecular visualization and cheminformatics. Py3Dmol is a Python wrapper for the JavaScript-based 3Dmol.js library, allowing users to create high-quality interactive 3D visualizations of molecular structures directly in Python environments, such as in Jupyter notebooks. It supports formats like PDB, MOL, and SDF providing tools to render molecular surfaces, labels, and animations, making it ideal for structural biology and computational chemistry.

RDKit is powerful tool for cheminformatics, enabling manipulation for chemical structures, computationa of molecular descriptors, and generation of 2D and 3D representations.It can also generate static and iinteractive visualizations, often in combinations with tools like Py3Dmol.

Syntax

```
See examples below
```

Examples

```
!pip install Py3Dmol
```

```
Collecting Py3Dmol
  Downloading py3Dmol-2.0.4-py2.py3-none-any.whl (12 kB)
Installing collected packages: Py3Dmol
Successfully installed Py3Dmol-2.0.4
```

```
import py3Dmol
```

```
p = py3Dmol.view(query='mmtf:1ylr')
```


```
p.setStyle({'cartoon': {'color':'spectrum'}})
```

```
!pip install rdkit
```

```
Collecting rdkit
  Downloading rdkit-2023.9.5-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (34.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 34.4/34.4 MB 42.8 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from rdkit) (1.25.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from rdkit) (9.4.0)
Installing collected packages: rdkit
Successfully installed rdkit-2023.9.5
```

```
from rdkit import Chem
from rdkit.Chem.Draw import IPythonConsole
from rdkit.Chem import Draw
from rdkit.Chem import rdAbbreviations
```

```
m = Chem.MolFromSmiles('COc1ccc(C(=O)[O-])cc1')
substructure = Chem.MolFromSmarts('C(=O)')
print(m.GetSubstructMatches(substructure))
m
```

Section 7: Classes

7.1 Building simple classes

Description

Python is an object-oriented high-level programming (OOP) language. Objects are an important part of Python programming. In fact, everything is an object in Python with default data and some functionality. Objects have attributes (variables) and methods (functions) specified in the class. To build an object, you need to use a constructor function (_ _ init _ _()). A constructor is a special function that is used to make an instance of a class. A class is like a blueprint that helps to create an object. You can then use that object (instance) of a class to assign some data and functionality. There are four main principles of object-oriented high-level programming.

1. Encapsulation,
2. Abstraction,
3. Inheritance, and
4. Polymorphism.

Note

Unlike variables, classes always start with the capital letters.

Syntax

```
class ClassName():
    def _ _ init _ _(self):
```

```
        block of code
```

Examples

```
# Creating your first class

class FirstClass:
      myVariable = 1
```

7.2 Classes and objects

Description

The *__init__()* function is called automatically everytime a new object is created. All python constructors must have at least one parameter, by convention it is called *self*, however, you can call it anything you want. The *self* parameter is used to access variables that belong to the class. First argument of the constructor is always filled automatically for us. Attributes are just variables inside of your constructor function.

Syntax

```
   class MyClass():
       def _ _init_ _ (self, parameter1, ...)
           self.parameter1 = parameter1
           .
           .
           .
```

Examples

```
# Using _ _init_ _() function with classes

# Class definition template
class Employee:

      # Class constructor
      def __init__(self, name, age):

        # Class attributes
        self.name = name
        self.age = age
```

```
# Using _ _init_ _() function with classes

# Class definition template
class Employee:
      def __init__(self, name, age):
        self.name = name
        self.age = age

      # Class method
      def listInfo(self):
        print("Hello, My name is", self.name, "and I am", self.age, "years old.")
```

```
# Using _ _init_ _() function with classes

# Class definition template
class Employee:
      def __init__(self, name, age):
        self.name = name
```

```
        self.age = age

      # Class method
      def listInfo(self):
        print("Hello, My name is", self.name, "and I am", self.age, "years old.")

# Creating a new object of a class
mySecondObject = Employee("John", 48)
```

```
# Using _ _init_ _() function with classes

# Class definition template
class Employee:
      def __init__(self, name, age):
        self.name = name
        self.age = age

      # Class method
      def listInfo(self):
        print("Hello, My name is", self.name, "and I am", self.age, "years old.")

# Creating a new object of a class, the first argument is the reference to the object.
mySecondObject = Employee("John", 48)

# Accessing the method of the class
mySecondObject.listInfo()
```

7.3 Encapsulation and abstraction

Description

Encapsulation involves the bundling of data (attributes) and method (functions) that operate on data within a single unit. A single unit is known as a class. It allows one to access and modify the attributes while keeping the attributes private.

Abstraction is a way to simplify the complex reality by creating a class and hide some unnecessary details. You can implement abstraction using classes and methods, where you show only the relevant details and hiding the internal implementation.

Syntax

```
    See examples below
```

Examples

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Adding properties of the class
        self.kind     = kind
        self.length   = length
        self.speed    = speed
```

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Properties of the class
        self.kind     = kind
        self.length   = length
        self.speed    = speed

    # Adding methods of the class
```

```
    def speed_up(self):
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

    def show_speed(self):
```

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Properties of the class
        self.kind     = kind
        self.length   = length
        self.speed    = speed

    # Adding methods of the class
    def speed_up(self):
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

    def show_speed(self):
        print("Current speed of my train is", self.speed)

# Creating an object
myTrain = Train("High-speed", 2, 0)
```

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Properties of the class
        self.kind     = kind
        self.length   = length
        self.speed    = speed

    # Adding methods of the class
    def speed_up(self):
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

    def show_speed(self):
        print("Current speed of my train is", self.speed, "miles an hour")

# Creating an object
myTrain = Train("High-speed", 2, 0)

myTrain.speed_up()
myTrain.speed_up()
myTrain.speed_up()

myTrain.show_speed()
```

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Properties of the class
        self.kind     = kind
        self.length   = length
        self.speed    = speed

    # Adding methods of the class
    def speed_up(self):
```

```
        self.speed += 1

    def speed_down(self):
        self.speed -= 1

    def show_speed(self):
        print("Current speed of my", self.kind, " train is", self.speed, "miles an hour")

# Creating an object
myTrain = Train("High-speed", 2, 0)


# Accessing the speed and kind properties directly
myTrain.speed = -200
myTrain.kind  = "Regional"
```

7.4 Private property

Description

One can have a property private that should be prefixed with an underscore _ or _ _ . While it does not make it entirely private, it sends a signal to a developer that these properties should not be accessed directly. Changing a name with two underscores (mangling) makes it harder to access or override. However, you wont be able to hide the attributes completely and you can still access them.

Syntax

```
     _myPrivateVariable

  _ _myProtectedVariable
```

Examples

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Properties of the class
        self.__kind       = kind
        self.length     = length

        # Making the speed property private
        self._speed    = speed

    # Adding methods of the class
    def speed_up(self):
        self._speed += 1

    def speed_down(self):
        self._speed -= 1

    def show_speed(self):
        print("Current speed of my", self.__kind, " train is", self._speed, "miles an hour")

# Creating an object
myTrain = Train("High-speed", 2, 0)

# Accessing the speed and kind properties directly
myTrain.speed = -200
myTrain.kind  = "Regional"

myTrain.show_speed()
```

7.5 Private method

Description

Protected and private methods are used to control access to class methods, by following naming convention rather than strict enforcement. A protected method marked with the single leading underscore (e.g. _method), it indicates that this method is designed to be used only within the class and subclasses. You can still access it from outside but it will be against the convention and should be avoided. A private method, denoted by two leading undescores (e.g. _ _method) undergoes name mangling, which remains it internally to include the class name (e.g. _ClassName__method), making it harder to access it from outside. This provides stronger indication of private method.

Syntax

```
      _myProtectedMethod()

    _ _myPrivateMethod()
```

Examples

```
class Train:

    # Constructor
    def __init__(self, kind, length, speed):

        # Properties of the class
        self.__kind       = kind
        self.length     = length

        # Making the speed property private
        self._speed    = speed

    # Adding methods of the class
    def speed_up(self):
        self._speed += 1

    def _speed_down(self):
        self._speed -= 1

    def show_speed(self):
        print("Current speed of my", self.__kind, " train is", self._speed, "miles an hour")

# Creating an object
myTrain = Train("High-speed", 2, 0)

# Accessing the private method speed_down.
myTrain.speed_down()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-31-890d820511f3> in <cell line: 0>()
     25
     26 # Accessing the private method speed_down.
---> 27 myTrain.speed_down()

AttributeError: 'Train' object has no attribute 'speed_down'
```

7.6 Enheritance

Description

Inheritance is a fundamental concept in object-oriented programming that allows a class, which is called a child or subclass, to inherit the attributes and methods from another class, called a parent or superclass. This promotes code reuse, modularity, and extensibility.

Inheritance is implemented by passing the parent class name in parentheses when defining the child class (e.g. class Child(Parent):) The Child class can use and override methods and properties from the Parent class. The Child class can define its own methods and properties. Python also supports multiple inheretence, where a clas can inherit from more than one Parent class.

Syntax

```
Parent class

  class ParentClass:
      def __init__(self):
        print("Parent constructor")

      def parentMethod(self):
        print("This is a method in the parent class")

Child class inheriting from ParentClass

  class ChildClass(ParentClass):
      def __init__(self):
        super().__init__()  # Call the parent class constructor
        print("Child constructor")

      def childMethod(self):
        print("This is a method in the child class")

Creating an object of the child class

myObject = ChildClass()
myObject.parentMethod()   # Inherited from ParentClass
myObject.childMethod()    # Defined in ChildClass
```

Examples

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
    pass

# Define a child or subclass class that inherits from Animal.
# Mammals are a specific class of animals, they are characterized
# by certain distinctive features like mammary glands and hair.

class Mammal(Animal):
    pass

# Define a child or subclass class that inherits from Mammal.
# Dog is the specific specie within the mammal class.

class Dog(Mammal):
    pass

#issubclass(Mammal,Animal)
#issubclass(Dog,Mammal)
#issubclass(Dog,Animal)

myDog = Dog()
print(myDog.__dict__)
```

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
    def __init__(self,name):
      self.name = name
```

```
# Define a child or subclass class that inherits from Animal.
class Mammal(Animal):
    pass

# If you do not specify the constructor explicitely in Dog subclass
class Dog(Mammal):
    pass

myDog = Dog("Rex")
print(myDog.__dict__)
```

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
    def __init__(self,name):
      self.name = name

# Define a child or subclass class that inherits from Animal.
# Mammals are a specific class of animals, they are characterized
# by certain distinctive features like mammary glands and hair.

class Mammal(Animal):
    def __init__(self, color):
      self.color = color

# If you do not specify the constructor explicitely in Dog subclass
# Python will look for it in superclass

class Dog(Mammal):
    pass

myDog = Dog("golden")
print(myDog.__dict__)
```

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
    def __init__(self,name):
      self.name = name

# Define a child or subclass class that inherits from Animal.
# Mammals are a specific class of animals, they are characterized
# by certain distinctive features like mammary glands and hair.

class Mammal(Animal):
    def __init__(self, name, color):
      Animal.__init__(self, name)
      self.color = color

# If you do not specify the constructor explicitely in Dog subclass
# Python will look for it in superclass

class Dog(Mammal):
    pass

myDog = Dog("Rex", "golden")
print(myDog.__dict__)
```

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
    def __init__(self,name):
      self.name = name

# Define a child or subclass class that inherits from Animal.
# Mammals are a specific class of animals, they are characterized
# by certain distinctive features like mammary glands and hair.
```

```
class Mammal(Animal):
    def __init__(self, name, color):
      super().__init__(name)
      self.color = color

# If you do not specify the constructor explicitely in Dog subclass
# Python will look for it in superclass

class Dog(Mammal):
    pass

myDog = Dog("Rex", "golden")
print(myDog.__dict__)
```

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
    def __init__(self,name):
      self.name = name

# Define a child or subclass class that inherits from Animal.
# Mammals are a specific class of animals, they are characterized
# by certain distinctive features like mammary glands and hair.

class Mammal(Animal):
    def __init__(self, name, color):
      super().__init__(name)
      self.color = color

# If you do not specify the constructor explicitely in Dog subclass
# Python will look for it in superclass

class Dog(Mammal):
    pass

myDog = Dog("Rex", "golden")
print(myDog.__dict__)

# Inheriting the name variable from Animal superclass
myDog.name = "Rocky"
print(myDog.__dict__)
```

Note

The *super()* keywords is used to call methods from a Parent or a Superclass withing a Child or Subclass. It is most commonly used to call the paent classes' constructor (_ _ *init* _ _) or to extend the functionality of inherited methods without fully overriding them. In case of *super()* keyword you do not need a self parameter since it is going to be taken automatically.

Syntax

```
class ParentClass:
    def __init__(self):
    print("Parent constructor")

class ChildClass(ParentClass):
    def __init__(self):
    super().__init__()  # Calls Parent's __init__
    print("Child constructor")
```

Examples

```
# Define a parent or superclass class.
# Animal is the broad category, encompassing all living organisms.

class Animal:
```

```
    message = "This message is from Animal superclass"
    def __init__(self,name):
      self.name = name

# Define a child or subclass class that inherits from Animal.
# Mammals are a specific class of animals, they are characterized
# by certain distinctive features like mammary glands and hair.

class Mammal(Animal):
    def __init__(self, name, color):
      super().__init__(name)
      self.color = color
      print(super().message)

# If you do not specify the constructor explicitely in Dog subclass
# Python will look for it in superclass

class Dog(Mammal):
    pass

myDog = Dog("Rex", "golden")
```

7.7 Method resolution order

Description

Method Resolution Order (MRO) is the order in which Python looks for a method or attribute when it is called on an instance of a class that involves inheritance. It becomes especially important in cases of multiple inheritance. When a class inherits from multiple paent classes, Python uses C3 linearization algorithm, also known as C3 MRO, to determine the order in which classes are searched.

Syntax

```
ClassName.mro()

ClassName._ _ mro _ _
```

Examples

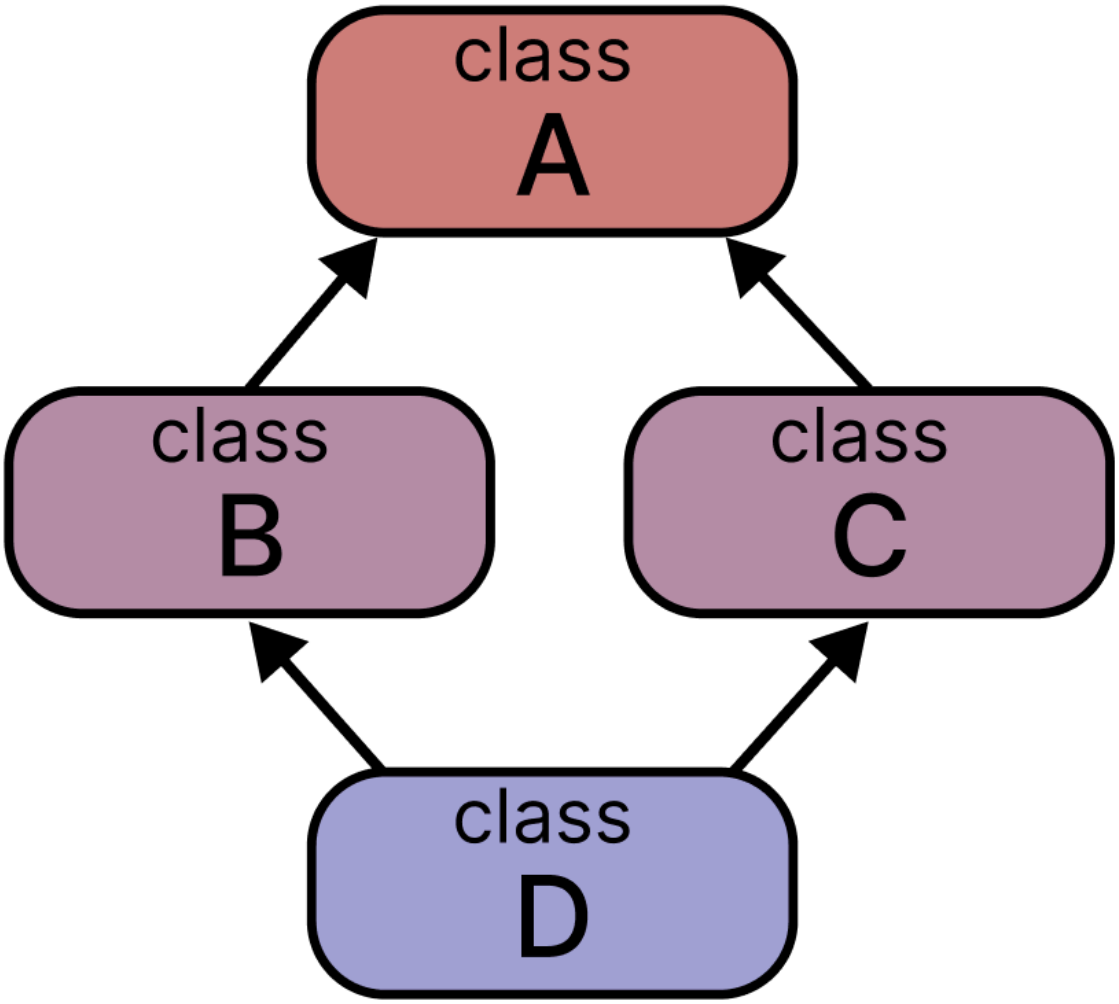
```
class Animal:
    pass

class Mammal(Animal):
    pass

class Canine(Mammal):
    pass

class Dog(Canine):
    pass

print(Dog.mro())   # returns a list

print(Dog.__mro__) # returns a tuple
```

7.8 Diamond problem

Description

There is a specific issue when you deal with multiple inheritance, it is called a diamond problem. A one class inherits from two classes that both inherits from a common superclass. This can call ambiguity in the Method Resolutin Order (MRO). Which method gets called when a method is invoked.

The Diamond Problem



Syntax

```
    See examples below
```

Examples

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Mammal(Animal):
    def speak(self):
        print("Mammal makes a sound")

class Feline(Mammal):
    def speak(self):
        print("Feline makes a sound: Meow, meow!")

class Canine(Mammal):
    def speak(self):
        print("Canine makes a sound: Woof, woof!")
```

```
class Cat(Feline, Mammal):
    pass

class Dog(Canine, Mammal):
    pass

# Create instances of Cat and Dog
myCat = Cat()
myDog = Dog()

# Call the speak method on both
myCat.speak()  # Which `speak()` will be called?
myDog.speak()  # Which `speak()` will be called?

# Let's check MRO

print(Cat.mro())
print(Dog.mro())
```

7.9 Method overriding

Description

Method overriding provides a subclass to define a method that has alredy being defined in its superclass. The subclass method overrides the method in the paren class, allowing the subclass to provide a specific method. You override method in superclass by using the same name of the method in subclass. When you call a method in subclass the overriden method is invoked.

Syntax

```
    See examples below
```

Examples

```
# Define a parent or superclass class.
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # This method will be overridden in subclasses

# Define a child or subclass class that inherits from Animal.
class Dog(Animal):
    def speak(self):
        print ("Woof, Woof!")

# Define a child or subclass class that inherits from Animal.
class Cat(Animal):
    def speak(self):
        print ("Meow, Meow!")

# Create instances of the subclasses
myDog = Dog("Rex")
myCat = Cat("Misty")

# Call the speak method on each instance
myDog.speak()
myCat.speak()
```

7.10 Polymorphism

Description

Polymorphism in object-oriented programming refers to the ability of different classes to provide a common interface for methods, allowing objects of different types to be treated as instances of a common superclass. It enables a single method to behave differently based on the object that invokes it. This allows for flexibility and reusability in code, as the same method can be used with different types of objects, and each type can implement its own version of the method. Polymorphism can be achieved through method overriding (where a subclass provides its own implementation of a method) or method overloading (where methods with the same name but different parameters exist). It is a core concept of object-oriented programming, enabling more dynamic and adaptable code.

Polymorphism is an ability for a method to be executed on different objects!

Syntax

```
    See examples below
```

Examples

```
# Define a parent or superclass class.
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # This method will be overridden in subclasses

# Define a child or subclass class that inherits from Animal.
class Dog(Animal):
    def speak(self):
        print ("Woof, Woof!")

# Define a child or subclass class that inherits from Animal.
class Cat(Animal):
    def speak(self):
        print ("Meow, Meow!")

# Create instances of Dog and Cat
myDog = Dog("Rex")
myCat = Cat("Misty")

# Create a function that takes an Animal object and makes it speak
# When myDog and myCat instances are passed to the function,
# it is showing polymorphism. The function works with objects of
# different classes and calls the appropriate speak method for
#each object.

def animalSpeak(animal):
    return animal.speak()

# Call the make_animal_speak function with different animal objects
animalSpeak(myDog)
animalSpeak(myCat)
```

Section 8: Exceptions

8.1 Syntax error

8.2 Value error

8.3 Exception with one error

8.4 Exception with three errors

8.5 Assertion exception

8.6 Exception with else statament

8.7 Exception with finally statement

8.8 Exception with raise statement

8.9 Exception object

8.10 Creating your own exceptions

Description

```
An exception is activated by an execution error that will terminate the
program execution. Exceptions are controlled by try and except branches.

1. try is executed
2. if no exception occurs, exception is skipped
3. if exception occures matching the exception name, the exception statement is executed
4. if there is no exception named it passes to the outer try statement

The list of most common exceptions in Python language:

ZeroDivisionError – raised if division by zero has attempted;
AttributeError – raised if method tried that does not exist;
SyntaxError – raised in violation of Python's grammar;
ValueError – raised when wrong use of values;
TypeError – raised when wrong type of data;
NameError – raised when variable or function does does not exist.
```

Syntax

```
  try:    block of code that causes an exception
  except: block of code to run when exception occures
```

Examples

```
try:
    # Trying to divide by zero
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Note

Exceptions are the errors that Python generates. Python does not know what to do and it raises an exception. One of the most often errors is the syntax error. Syntax errors are very easy to find and fix. Here's an example of simple syntax error.

8.1 Syntax error

Description

A SyntaxError in Python occurs when the code does not follow the proper structure or grammar rules of the Python programming language. It typically arises when there is a mistake in the code's formatting, such as missing punctuation (e.g., parentheses, colons), misspelled keywords, incorrect indentation, or improper use of operators. SyntaxErrors are detected by the Python interpreter when the program is compiled, and the interpreter will stop execution, pointing out the location where the error occurred. For example, omitting a colon at the end of an if statement or failing to close a parenthesis will trigger a SyntaxError.

The SyntaxErrors are the most common errors in programming!

Syntax

```
  There is no defined syntax for the Syntax Error
```

Examples

```
if True;
print("Hello, World!")
```

8.2 Value error

Description

A ValueError occurs when a function receives an argument of the correct type but an inappropriate value that cannot be processed. This Error typically arises when an operation or function call receives an argument that is valid in terms of its data type but not valid for the intended operation. For example, trying to convert a string that does not represent a number into an integer using *int()* will raise a ValueError. An example would be int("abc"), which will fail because the string "abc" cannot be interpreted as an integer. This type of error highlights a mismatch between the value provided and the expected value for a particular operation.

Syntax

```
There is no defined syntax for the Value Error
```

Examples

```
# Exception for division by zero

myDivision = (1.0/float(input("Enter any number except zero ")))
print(myDivision)
```

```
try:
  myDivision = (1.0/float(input("Enter any number except zero ")))
  print(myDivision)
except:
  print('Wrong input')
```

```
try:
  myDivision = (1.0/float(input("Enter any number except zero ")))
  print(myDivision)
except ValueError:
  print('Wrong value input')
except ZeroDivisionError:
  print('Division by zero')
except TypeError:
  print('Error')
```

Note

When you see the error you are given some useful information, a line that produces the error, name of the error, and you can see a descriptive message that will help you to debug the program.

8.3 Exception with one error

Description

This example demonstrates the use of a *try: except:* block to handle a ZeroDivisionError in Python. The ZeroDivisionError is raised when a division or modulo operation is performed with zero as the denominator, which is mathematically undefined. By using a *try* block, we can attempt to perform a division operation, and if the denominator is zero, the program will raise the ZeroDivisionError. The *except* block

catches the error, preventing the program from crashing, and allows the programmer to display a custom error message or handle the situation in a more controlled manner.

This approach ensures the program continues running even when such an error occurs.

Syntax

```
try:
    # Code that might raise a ZeroDivisionError
    expression()

except ZeroDivisionError:
    # Code to handle the ZeroDivisionError
    print("Error: Cannot divide by zero.")
```

Examples

```
# Exception for division by zero

def devisionByZero():
   myDivision = (1.0/float(input("Enter any number except zero ")))
   return myDivision

try:
  devisionByZero()
except ZeroDivisionError:
  print('Division by zero has been attempted')
```

8.4 Exception with three errors

Description

This example demonstrates the use of a *try: except:* block to handle multiple types of exceptions: ValueError, ZeroDivisionError, and TypeError. By specifying different *except* blocks for each exception type, the program can handle various errors in a more controlled way. For example, a ValueError may be raised if the input is not a valid number, a ZeroDivisionError occurs when attempting to divide by zero, and a TypeError is triggered when an operation is performed on incompatible data types. This approach allows the program to catch specific errors and provide meaningful error messages, making it easier to debug and maintain.

Syntax

```
try:
    # Code that might raise exceptions
    expression()

except Error1:
    # Handle Error1
    print("Message1")

except Error2:
    # Handle Error2
    print("Message2")

except Error3:
    # Handle Error3
    print("Message3")
```

Examples

```
# Exception for division by zero

def devisionByZero():
   myDivision = (1.0/float(input("Enter any number except zero ")))
   return myDivision

try:
    devisionByZero()
except ValueError:
  print("Wrong value input")
except ZeroDivisionError:
  print("Division by zero")
except TypeError:
  print("Wrong type")
```

8.5 Assertion exception

Description

The *assert* keyword is used as a debugging aid that tests a condition as a sanity check while the program is running. If the condition evaluates to True, the program continues normally. However, if it evaluates to False, Python raises an AssertionError exception, optionally with a custom message. This is useful during development to catch unexpected conditions early and ensure that the code behaves as expected. Assertions are typically used to verify internal states, inputs, or outputs, and they can be disabled globally when Python is run in optimized mode (using the -O flag), making them a lightweight and effective debugging tool.

The assertion is uset for debugging and documenting your code!

Syntax

```
    assert condition, "Optional error message"
```

Examples

```
def inverseFunction(a):
  assert (a > 0), 'Division by zero, attempt!'
  return 1/a
```

```
print(inverseFunction(0.0))
```

```
for subclass in ZeroDivisionError.__subclasses__():
    print(subclass.__name__)
```

8.6 Exception with else statement

Description

The *else* block in a *try: except: else:* structure is executed only if no exceptions occur in the try block. This allows you to separate the code that should run only when everything goes smoothly from the code that handles exceptions. If an Error is raised and caught by an *except* block, the else block is skipped. Using *else* makes your code cleaner and easier to read by clearly distinguishing between Error handling and successful execution paths. It's particularly useful when you want to run additional logic only when the try block succeeds without errors.

Syntax

```
try:
    # Code that might raise an exception
    expression()

except ExceptionType as e:
    # Code to handle the exception
    print("An error occurred:", e)

else:
    # Code that runs if no exception occurred
    print("Operation successful.")
```

Examples

```
try:
  my_number = int(input("Enter any number"))
except:
  print('You did not provide a number')
else:
  print('Great, this number works fine')
```

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Division by zero is not allowed.")
else:
    print("The result is:", result)
```

8.7 Exception with finally statement

Description

The *finally* keyword in Python is used at the end of a *try: except:* block to define a block of code that will always be executed, regardless of whether an exception was raised or caught. This is especially useful for cleanup tasks such as closing files, releasing resources, or resetting variables, ensuring that essential final steps are performed no matter what happens in the try or except blocks. The *finally* block runs even if an exception is not caught, or if a *return* or *break* statement is executed earlier in the *try* or *except* section. By placing finally at the end of the structure, it guarantees reliable execution of critical code regardless of how the preceding blocks behave.

Syntax

```
try:

    expression()

except ExceptionType as e:
    print("An error occurred:", e)

finally:
    expression() # It will be always executed
```

Examples

```
try:
  my_number = int(input("Enter any number"))
except:
  print('You did not provide a number')
else:
  print('Great, this number works fine')
```

```
finally:
  print('Great, it finally works')
```

8.8 Exception with raise statement

Description

The *raise* keyword is used in Python to manually trigger exceptions, allowing you to intentionally stop program execution when certain conditions are met. This is especially useful during development to debug and verify your code by ensuring that specific rules or constraints are enforced. For instance, you might want to raise a Value Error if a function receives a negative number when only positive values are allowed. By inserting raise at the right place in your code, you can generate meaningful exceptions that help identify logic errors early and improve code robustness.

Syntax

```
raise ExceptionType("Custom error message")
```

Examples

```
try:
  raise Exception
except Exception as e:
  print(e.args)
```

```
try:
  raise Exception ('I like it')
except Exception as e:
  print(e.args)
```

```
def myNumber(a):
  try:
    return 1/a
  except:
    print('Wrong input')
```

```
myNumber(0)
```

```
def myNumber(a):
  try:
    return 1/a
  except:
    print('Wrong input')
    raise
```

```
myNumber(0)
```

```
def checkNumber(x):
    if x < 0:
       raise ValueError("Negative numbers are not allowed.")
    print("Number is valid.")

checkNumber(5)    # Valid
checkNumber(-2)   # Raises ValueError
```

8.9 Exception object

Description

This example demonstrates the use of a *try: except:* block with objects to handle errors gracefully and provide meaningful error messages. When performing operations involving objects, such as accessing methods or attributes, errors like AttributeError, TypeError, or ValueError may occur. By enclosing the potentially problematic code within a try block and handling specific exceptions in the except block, the program can catch and respond to these issues without crashing. Additionally, using the exception object (e.g., except Exception as e:) allows the program to print or log the exact error message, giving clear insight into what went wrong, which helps with debugging and improving the user experience.

Syntax

```
try:

    expression()

except Exception as e:

    print("An error occurred:", e)
```

Examples

```
def compare_two_numbers(a,b):
    if a > b: return a
    else: return b
```

```
try:
      compare_two_numbers(5,'d')
except Exception as e:
       print (e)
```

```
try:
   raise Exception ('I like it')
except Exception as e:
  print(e.args)
```

```
try:
   5 < 'a'
except Exception as e:
  print(e.args)
```

```
try:
   1/0
except Exception as e:
  print(e.args)
```

8.10 Hierarchy of exceptions

Description

The hierarchy of exceptions in Python is organized as a class-based tree structure, with the base class BaseException at the top. All built-in exceptions are derived from this base class, either directly or indirectly. The most commonly used superclass for user-defined exceptions is Exception, which inherits from BaseException. Under Exception, there are many specific exception classes such as ValueError, TypeError, IndexError, and others, each representing different types of errors. Python also includes other branches under BaseException, such as SystemExit, KeyboardInterrupt, and GeneratorExit, which are used for system-level events and are typically not caught in most programs. Understanding this hierarchy allows developers to write more precise try-except blocks by catching general or specific exceptions as needed.

```
There is an hierarchy of exceptions.

      BaseExceptions
```

```
            |
          Exception - SystemExit - KeyboardInterrupt - ...
            |
          ArithmeticError - LookupError - TypeError - ValueError _ ...
            |                 |
          ZeroDivisionError  IndexError - ...
            |                 |
          ...               KeyError
```

Syntax

```
    print(subclass._ _ name _ _ )
```

Examples

```
for subclass in BaseException.__subclasses__():
    print(subclass.__name__)
```

```
for subclass in Exception.__subclasses__():
    print(subclass.__name__)
```

```
for subclass in ArithmeticError.__subclasses__():
    print(subclass.__name__)
```

LookupError

Description

A LookupError is a base class for exceptions that occur when a key or index is not found. It includes two common subclasses: KeyError and IndexError. These are typically raised when you're trying to access something that doesn't exist, like a key in a dictionary that isn't present or an index out of range in a list.

Syntax

```
    See examples below
```

Examples

```
# Example with dictionary (KeyError)

myDict = {'apple': 10, 'banana': 5, 'cherry': 7}

try:
    # Trying to access a key that doesn't exist
    print(myDict['orange'])
except KeyError as e:
    print(f"KeyError: The key '{e}' was not found in the dictionary.")
```

```
# Example with list (IndexError)

myList = [1, 2, 3, 4]

try:
    # Trying to access an index that is out of range
    print(myList[5])
except IndexError as e:
    print(f"IndexError: The index {e} is out of range for the list.")
```

```
# Example handling both KeyError and IndexError as LookupError

myDict = {'apple': 10, 'banana': 5, 'cherry': 7}
myList = [1, 2, 3, 4]

try:
    # Accessing a dictionary key that doesn't exist
    print(myDict['orange'])

    # Trying to access an out-of-bounds list index
    print(myList[5])
except LookupError as e:
    print(f"LookupError: {e}")
```

8.11 Creating your own exceptions

Description

Creating your own exceptions in Python involves defining a custom exception class that inherits from the built-in Exception class or one of its subclasses. This allows you to raise errors that are specific to your application's logic, making it easier to handle and debug unexpected situations. To create a custom exception, you define a class using the class keyword, give it a name that typically ends in "Error" (e.g., MyCustomError), and optionally customize the message or behavior by overriding methods like (_ _ *init* _ _). You can then use the raise keyword to trigger your custom exception when a certain condition is met. For example, raise MyCustomError("Something went wrong") will stop execution and display the custom message, which can be caught using a try-except block.

Syntax

```
class MyCustomError(Exception):
    def _ _init_ _(self, message):
    super()._ _init_ _(message)
    .
    .
    .
raise MyCustomError("Message.")
```

Examples

```
# Define a custom exception
class DivisionByOddNumberError(Exception):
    def __init__(self, message="Cannot divide by an odd number."):
        self.message = message
        super().__init__(self.message)

# Function that uses the custom exception
def divideByEvenNumber(x, y):
    if y % 2 != 0:
        raise DivisionByOddNumberError()
    return x / y

# Test the function
try:
    result = divideByEvenNumber(10, 3)
    print("Result:", result)
except DivisionByOddNumberError as e:
```

```
    print("Custom Exception Caught:", e)
```

Section 9: Input and output (I/O) operations

9.1 User input

9.2 File input and output

9.3 Creating a new file

9.4 Open file with "with" option

9.5 Print function

9.6 Formatting output from print function

Note

Input and output in Python involves reading data from various sources and writing data to various destinations. Pyhton provides several ways to handle input and output. Here are examples of the most commpn methods.

9.1 User input

Description

User input is allowed in Python. The *input()* method (python 3.6) or raw_input method in python (2.7) can be used.

Syntax

```
variable = input( " Corresponding text for a user " )
```

Examples

```
# The user input

userName = input("Enter your user name plz: \n")

print("Your username is: " + userName)
```

9.2 File input and output

Description

File input and output in Python is a way to read from and write to a file on your loacal machine. Python provides a few methods to work with files. Files in Pyhton can be created, read, appended, ... open() function can be used to work with files in Python. This function takes two parameters: filename and mode.

```
Method (mode) for opening a file:
r - Read - Default value. Opens a file for reading, error if the file does not exist
a - Append - Opens a file for appending, creates the file if it does not exist
w - Write - Opens a file for writing, creates the file if it does not exist
x - Create - Creates the specified file, returns an error if the file exists

In the text or binary modes
```

```
t - Text - Default value. Text mode
b - Binary - Binary mode (e.g. images)
```

Syntax

```
file = open("filename", "mode")
```

Examples

```
# To open a file - requires explicit closing

myFirstFile = open( "myDataFile.txt", "rt" )
myFirstFile.close()                      #Close the file when you are finished.
```

```
# To open a file - requires explicit closing

myFirstFile = open( "myDataFile.txt", "r" )

myFirstFile.read()
myFirstFile.readline()

myFirstFile.close()                      #Close the file when you are finished.
```

```
# Loop through the entire file

myFirstFile = open( "myDataFile.txt", "r" )

for i in myFirstFile: print(i)

myFirstFile.close()                     # Close the file when you are finished.
```

```
# To append to a file you need to use "a" parameter

myFirstFile = open( "myDataFile.txt", "a" )

myFirstFile.write("It is going to the end of a file")

myFirstFile.close() #Close the file when you are finished.

# Open and read the file after the appending

myFirstFile = open( "myDataFile.txt", "r" )

print(myFirstFile.read())

myFirstFile.close() #Close the file when you are finished.
```

9.3 Creating a new file

Description

Creating your own file in Python is done using the built-in *open()* function with the appropriate mode. To create and write to a new file, you can use mode 'w' (write) or 'x' (exclusive creation), where 'w' creates the file if it doesn't exist or overwrites it if it does, while 'x' raises an Error if the file already exists. After opening the file, you can use the *write()* method to insert content into it, and it's important to close the file afterward using *close()*, or better yet, use a with block to handle the file automatically. For example, with open("example.txt", "w") as f: creates a file named example.txt and opens it for writing, ensuring it is properly closed after the block ends.

```
Method (mode) for opening a file:
r  - Open   - will open a file for reading;
x  - Create - will create a file, returns an error if the file exist;
```

```
a  - Append - will create a file if the specified file does not exist;
w  - Write  - will create a file if the specified file does not exist.
```

Syntax

```
file = open("filename.txt", "w")
file.write("Your content goes here")
file.close()
```

Examples

```
# Creating a new file

myFirstFile = open( "myDataFile2.txt", "x" ) # a new and empty file is created
myFirstFile = open( "myDataFile3.txt", "w" ) # if there is no file, it will be created
myFirstFile = open( "myDataFile4.txt", "a" ) # if there is no file, it will be created

myFirstFile.close()
```

```
# To delete the file you need to import the os module

import os

if os.path.exists("myDataFile.txt"):
   os.remove("myDataFile.txt")
else:
  print("This file does not exist")
```

```
# To delete the entire folder

import os

os.rmdir("myFolder")
```

9.4 Open file with "with" option

Description

Using the open() method requires you to use the close() method to close the file. "with" option allows to open files as it returns a file object, which has methods and attributes to manipulate the files.

Syntax

```
with open("filename.txt", "w") as file:
    file.write("Your content goes here")
```

Examples

```
# Using "with" - no explicit closing of a file is required

with open("myDataFile.txt", "a") as myFirstFile:
  myFirstFile.write("Hello World!")

# You do not need to close it explicitely
```

9.5 Print function

Description

The *print()* function in Python is used to display output to the console. It is one of the most commonly used built-in functions and can print strings, numbers, variables, and even the results of expressions. By default, *print()* separates multiple items with a space and ends the output with a newline character, but this behavior can be customized using the sep and end parameters.

Print function is usually used for to print python objects as standard output!

```
objects - strings, tuples, lists, etc.
sep - an optional parameter that is used to define a separation.
end - an optional parameter that is used to define an end of a string printed.
file - an optional parameter when writing to the file.
flush - an optional boolean parameter setting flushed or buffered output.
```

Syntax

```
print(objects, sep, end, file, flush)
```

Examples

```
# Printing the objects

myList   = [ 1, 2, 3, 4, 5 ]

myTuple  = ( 1, 2, 3, 4, 5 )

myString = "1, 2, 3, 4, 5"

print(myList, myTuple, myString)
```

```
[1, 2, 3, 4, 5] (1, 2, 3, 4, 5) 1, 2, 3, 4, 5
```

```
# Printing objects with a separator

myList   = [ 1, 2, 3, 4, 5 ]
myTuple  = ( 1, 2, 3, 4, 5 )
myString = " 1, 2, 3, 4, 5 "

print(myList, myTuple, myString, sep="/")
```

```
[1, 2, 3, 4, 5]/(1, 2, 3, 4, 5)/ 1, 2, 3, 4, 5
```

```
# Printing objects with a separator and an end

myList   = [ 1, 2, 3, 4, 5 ]
myTuple  = ( 1, 2, 3, 4, 5 )
myString = " 1, 2, 3, 4, 5 "

print(myList, myTuple, myString, sep=" ; ", end=" END")
```

```
[1, 2, 3, 4, 5] ; (1, 2, 3, 4, 5) ;  1, 2, 3, 4, 5  END
```

```
# Using flush command

import time
countSeconds = 3

for i in reversed(range(countSeconds + 1)):
  if i> 0:
    print(i, end="/", flush=True)  #flush specifies if it is flushed (True) or buffered (False)
    time.sleep(1)
  else:
    print('End of a file')
```

9.6 Formating output from print() function

Description

Formatting output using the *print()* function in Python allows you to display information in a structured and readable way. This can be done using the modulus operator (%), the *format()* method, and the f-strings (formatted string literals).

The modulus operator (%) for string formatting is an older method in Python that uses placeholders like %s, %d, and %f to insert values into a string. The % operator is placed after the string, followed by the value(s) to be inserted. Common format specifiers include %s for strings, %d for integers, and %.2f for floating-point numbers with two decimal places. This method is similar to C-style formatting and still works in modern Python.

The *format()* method works similarly with placeholders: print("Name: {}, Age: {}".format(name, age)).

F-strings *f'{}'* are the most modern and preferred method, introduced in Python 3.7, and allow expressions to be embedded directly in string literals using curly braces, like print(f"Name: {name}, Age: {age}"). You can also control the alignment, width, number formatting, and padding of values, making it useful for printing tables, reports, or aligned data. This flexibility makes formatted printing a powerful tool for creating clean, professional output.

Syntax

```
       ' %d %s ... %.2f '    %         ( 1, 2, ..., n)
      |------------------|   %   |-------------------------|
            template      modulus    positional arguments
print(' %d %s ... %.2f'     %         ( 1, 2, ..., n)     )


     |------------------|.format|-------------------------|
            template       format    positional arguments
print(' {} {} ... {} n'  .format      ( 1, 2, ..., n )    )

print(' {key} ... {} n'  .format     (key = value, ...)   )



print(f'{} {} ... {} n' )
```

Examples

```
# Output formatting in python
# This program is showing how to use
# string modulo operator(%) to print various data types

# print integer and float value
print("Integer number : %2d, float number : %5.2f" % (11, 05.333))

# print integer value
print("Total students : %3d, Boys : %2d" % (240, 120))

# print octal value
print("%7.3o" % (25))

# print integer value
print("%7.3d" % (25))

# print exponential value
print("%10.3e" % (356.08977))
```

```
# Output formatting in python
# This program is showing how to use
# string modulo operator(%) to print various data types
print('%d %s cost $%.2f' % (3, 'oranges', 5.11))
```

```
# Control output using format() method with one argument

myString = "Python language was created in {}"

print(myString.format(1985))
```

```
# Python's string.format() method is used instead
print('{0} {1} cost ${2}'.format(3, 'oranges', 5.11))
#     |-----------------|.format|-------------------------|
#           template     .format    positional arguments
```

```
# Using keyword parameters instead of positional arguments
print('{quantity} {item} cost ${price}'.format(quantity=3,item='oranges',price=5.11))
```

```
# Output formatting in Python
# This program is showing how to use
# f-string to print various data types

quantity = 3
item = 'oranges'
price = 5.11

print(f'{quantity} {item} cost ${price}')
```

Section 10: Miscellaneous

10.1 Closure function

Description

A closure function in Python refers to a function that retains access to variables from its enclosing scope, even after that scope has finished execution. This concept is particularly useful when you need to remember the state across multiple calls to a function. In a closure, the inner function has access to the variables of the outer function, allowing it to retain and modify the state as needed. This ability to "remember" the outer variables is what makes closures powerful. Closures are often used for tasks like data encapsulation, maintaining a running total or counter, and implementing decorators. They provide a way to keep track of data in a compact, efficient manner without needing to use global variables or objects.

Syntax

```
def outer_function(outer_variable):
    def inner_function(inner_variable):
        # The inner function can access the outer variable
        return outer_variable + inner_variable
    return inner_function
```

Examples

```
def adder(base_value):
    def add(new_value):
        return base_value + new_value
    return add
```

```
# Create a closure that adds 10 to any number
add_10 = adder(10)

print(add_10(5))   # Output: 15
print(add_10(20))  # Output: 30

# Create another closure that adds 100
add_100 = adder(100)

print(add_100(50))  # Output: 150
```

```
15
30
150
```

```
def multiplier(factor):
    def multiply(value):
        return factor * value
    return multiply

# Creating a closure with a factor of 3
multiply_by_3 = multiplier(3)

# Now using the closure to multiply different values
print(multiply_by_3(10))  # Output: 30
print(multiply_by_3(5))   # Output: 15
```

```
30
15
```

10.2 Map function

Description

The *map()* function in Python is a built-in functional programming tool used to apply a given function to every item in an iterable (like a list, tuple, or set) without writing an explicit loop. It's especially useful for transforming large datasets efficiently and in a clean, readable way.

Syntax

```
map(function, iterable)
```

Examples

```
# Example with regular function

def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))

print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

```
# Example with lambda function

numbers = [10, 20, 30]
doubled = list(map(lambda x: x * 2, numbers))

print(doubled)  # Output: [20, 40, 60]
```

Note

If working with very large files or datasets, consider combining *map()* with *iter()* on file objects, or use it with *multiprocessing.Pool.map()* to parallelize the transformation across multiple CPU cores for performance boosts.

10.3 Filter function

Description

The *filter()* function is a built-in Python tool used to filter elements from an iterable based on a condition defined in a function. It returns only the items that make the function evaluate to True. It's commonly used when working with large datasets to selectively keep elements without writing verbose loops.

Syntax

```
filter(function, iterable)
```

Examples

```
# Example using modulus operator %

def is_even(n):
    return n % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(is_even, numbers))

print(even_numbers)  # Output: [2, 4, 6]
```

```
# Example using lambda function

numbers = [10, 15, 20, 25, 30]
filtered = list(filter(lambda x: x > 20, numbers))

print(filtered)  # Output: [25, 30]
```

10.4 Reduce function

Description

The *reduce()* function is part of the functools module and is used to apply a function cumulatively to the items of an iterable, reducing it to a single value. It's commonly used for operations like summing, multiplying, or combining elements in a sequence. This function is ideal for processing large datasets where you need a final aggregated result, such as total, max, min, or a single combined string.

Syntax

```
from functools import reduce

reduce(function, iterable[, initializer])
```

Examples

```
# Example using regular function

from functools import reduce

def multiply(x, y):
```

```
    return x * y

numbers = [1, 2, 3, 4]
product = reduce(multiply, numbers)
```